1-1-2005

Visual programming for virtual reality application development

Zheng Min
Iowa State University

Visual programming for virtual reality application development

by

Zheng Min

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Julie Dickerson
Patrick Patterson

Iowa State University

Ames, Iowa

2005

Graduate College

Iowa State University

This is to certify that the master's thesis of

Zheng Min

has met the thesis requirements of Iowa State University

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. Carolina Cruz-Neira for her guidance and patience. This work would not have been possible without her support and encouragement. I would also like to thank the VR Juggler development team for their excellent work and their helps on my development. A special thanks is given to Patrick Hartling for enduring my endless questions. Finally I would like to thank my wife Jiang for always believing in me and supporting me even when I was unsure.

ABSTRACT

This thesis presents VRAE (Virtual Reality Application Editor), a visual programming tool created over the VR Juggler toolkit, to enable application developers to concentrate on domain-level issues rather than VR technical details. Thus the development of VR applications can be greatly simplified. The main research focus is on applying visual programming and code generation technology to VR application development. The goal is to construct a development framework to enable reusing of existing VR application development experience, shorten the time that it takes to develop applications, and allow VR users with limited programming experience to create their own applications.

In addition to providing a development structure, this framework offers the extensibility for adding new application "types" in the future. Because the work relies on VR Juggler as the underlining tool, it addresses application design issues such as computer platform independency, and display compatibility.

CHAPTER 1 INTRODUCTION

Motivation

For the past decade, Virtual Reality (VR) has evolved at an amazing speed to the point that it has been accepted by academic and industry groups. It has been used to advance engineering appliance in widespread areas, from scientific visualization, CAD, vehicle simulation to entertainment. Empowered by the rapid development of the computer hardware and software, VR technology provides engineers, scientists and many other users with a revolutionary method to solve problems in various application domains. Compared to traditional human-computer-interaction technology, VR allows users to handle their data more naturally and to gain deeper insight into their domain problems.

The computer hardware capability and the software efficiency are two key issues empowering the VR technology. However, the extent of benefits of VR relies more on how easily people in different domains can take advantage of this technology, in other words, the ability of quickly developing VR applications. No matter how powerful the VR facilities could be, to be practical, they must be able to serve more people in a great range of domains.

However, Virtual Reality systems are highly complicated hardware/software systems. Heterogeneous components are integrated into one VR system, making the configuration of such a system extremely hard. Additionally, VR applications have higher real-time performance requirement than common computer applications. The system complexity and time-critical performance requirements make the development of a VR application much more involved than that of a conventional interactive computer application.

Fortunately, research groups have produced a wide range of software libraries to help users to handle the system complexity. These software libraries hide the complexity of VR hardware from developers of VR applications. Thus, they only need to deal with a set of Application Program Interfaces (API). Examples of these libraries are the CAVELibs (Cruz-

Neira, 1995), DIVERSE (DIVERSE), MR Toolkit (Shaw, 1993), and WorldToolKit (Sense8, 1998) etc. Among these tools, and of particular interest in this research is VR Juggler (Bierbaum, 2000) developed in Virtual Reality Application Center (VRAC) at Iowa State University (ISU), perhaps one of today's most popular toolkits for VR. Working with these software libraries, software engineers and other engineers with sufficient programming skills can build complex VR applications. Numerous applications in a variety of areas have been developed based on these libraries. Engineers, scientists and artists have greatly benefited from these applications (VRAC).

However, the majority of engineers, scientists and artists do not have sufficient programming skills required to build a VR application. It is hard or even impossible for them to develop VR applications on their own. However, their knowledge and skills in their own domains are of great value to solve real world problems or to create fantastic art works. Sometimes, they can get help from software engineers. However, it is not always the case. To them, it would be nice if the VR application development could be much easier and intuitive. Moreover, as the technology developing, more and more persons with diverse skills will get involved in VR applications design. There are strong demands on novel easy-to-use tools for VR application development.

On the other hand, even for experienced programmers, building up a VR application still needs significant time and effort. Furthermore, a big portion of this work actually has been performed repetitively. A fast application prototyping and configuration tool can help them avoid reinventing-the-wheel and give them more time to focus on solving domain level problems.

Up to date, there have been many efforts to provide tools to ease applications development. Many open source or commercial VR authoring software packages have been built, such as EON Studio provided by EON Reality (EON), Virtools Dev developed by

Virtools (Virtools), WorldUp by Sense8 (Sense8), and Alice (Stage3). These tools simplify the VR application development. However, they have their own limitations, as elaborated in a later chapter of this thesis. There is still a need for a good environment to quickly develop powerful VR applications.

Statement of Purpose

The research presented in this thesis focuses on applying end-user programming and code generation technology to facilitate the development of VR applications. The goal of the research is to construct a development framework to enable reusing of existing VR application development experience. With the resulting tool, application developers can create and connect with domain level application components in a way that does not require advanced software engineering skills. Thus the development of VR applications will be greatly simplified.

To achieve this goal, the research presented in this document analyzes the common requirements for Virtual Reality application development and studies the code generation technology, as well as different kinds of end-user programming, in particular virtual programming technology. Based on the analysis, this research provides an application development framework and finally implements a visual programming tool for VR application development.

Scope of Work

The research in this thesis has been structured in several stages, which are listed as following:

1. **Survey of current Virtual Reality applications**

To date, a great number of Virtual Reality applications have been developed or are under-development. These applications cross a variety of domains. There have been quite some mature solutions or architectures that can be reused by other developers. Analyzing and

categorizing existing VR applications is a natural step toward finally constructing a development framework. Chapter 2 covers the result of the survey.

2. **Study techniques that enable end-user programming**

End-user programming is a research area that focuses on providing programming languages or developing tools to non-experts of computers. The methods of end-user programming, such as visual programming, and domain-specific languages, have been proved good means to flatten the steep learning curve of software development thus to speed up the progress. Furthermore, another technology, code generation, is also needed for building up complete VR applications. Proper techniques of these two areas are studied to support design and implementation decisions in this research. This content is also covered in Chapter 2.

3. **Analyze requirement of VR applications development**

The requirements of VR application development, especially of simplifying the development complexity, are defined based on the study of existing VR applications. According to the purpose of this research, more focus has been placed on how to simplify or to speed up the development. The development of a VR application can be divided into many subtasks. Some of them, such as geometrical modeling, can be simplified by applying individual off-the-shelf software. While other subtasks, such as designing the interaction functionality, are the main topics to be addressed in this research. Chapter 3 presents the result of this research stage.

4. **Analyze existing VR development tools**

Up until now, many researchers and developers have made significant effort to improve VR application development. Both open source and commercial tools are emerging. This

research has studied three commercial tools and three academic research works. The result provides more ideas of the direction of this research. In Chapter 4, the analysis is described.

5. **Design a framework for simplifying the VR development**

The research of VR applications and the development procedure shows that identical or similar components exist in different VR applications. Moreover, the development procedures of many applications contain similar sequences of actions. Starting from the similarity, this research works on constructing an underneath framework for application development tools. In addition to providing a development structure, this framework offers the extensibility for adding new application models in the future. The system design detail is given in Chapter 5.

6. **Implement a visual programming tool for VR application development**

Founded on the framework constructed in Stage 5, a Java-based program is developed as a starting point of providing practical development tools. This software is able to prove the main concepts of the system design, and to present valuable evaluation information. Future research directions are suggested by analyzing the result of this implementation. The design detail of the tool is elaborated in Chapter 6. Moreover, usage and results are discussed in the following chapter. Conclusions and future works are covered in Chapter 8.

CHAPTER 2 BACKGROUND

Virtual Reality Applications

Virtual reality

Virtual reality (VR) has been used in a wide variety of context, some times miss used and confused with other technologies such as Web 3D, computer animation, and even special effects in movies. For the context of this document, we use the term virtual reality and virtual environments referring to Rory Stuart's definition (2001): "[Virtual environments are] systems capable of producing an interactive immersive multisensory 3-D synthetic environment; it uses position-tracking and real-time update of visual, auditory, and other displays (e.g., tactile) in response to the user's motions to give the users a sense of being 'in' the environment, and it could be either a single or multiuser system." (Stuart, 2001)

This definition points out three essential characteristics of VR technology: interaction, immersive and computer-generated.

Interaction in the context of VR means that the user-computer interaction is highly emphasized. VR has the potential to provide users with a novel (and potentially more natural) way to interact with computers than conventional WIMP interfaces typically found in desktop systems. For example, in VR users may directly "grab" and "manipulate" objects by "hand" using a tracked data glove, or they can control their location in the virtual space simply with changing their posture.

Being computer-generated (or synthetic) implies that a virtual environment presents its user with a three-dimensional model of a scene. People may think this scene as a virtual "world". This "world" could contain buildings, plants, vehicles like in the real world; it could also be full of symbols of fluid flow or molecules that cannot be directly seen in the real world.

Being immersive indicates that users of the VR technology have feeling of being "inside" of the synthetic "world". Multisensory channels are involved to make users not only "see" and "hear" the data, but also "feel" it.

In summary, these attractive features entail a complicated hardware structure with tight real-time requirement. Interaction and immersion are supported by heterogeneous devices and equipment, such as data gloves, position-tracking systems, a vehicle cab, and 3D stereo rendering systems, e.g. HMD (Bungert, 2005) CAVE (Cruz-Neira, 1993), ImmersaDesk (FakeSpace), and PowerWall (PowerWall). Their presence in turn increases the system complexity and brings challenging application implementation issues with them. On the other hand, performance of the computer system cannot afford being sacrificed, --with a poor system response, whatever novel hardware device is used, users' feeling of natural interaction and immersion will be severely impacted (Held, 1991).

Virtual reality applications

Virtual reality applications are programs that utilize VR technology to solve practical problems. These programs take advantage of the natural human-computer interaction provided by VR systems, giving users the opportunity to explore the data that is otherwise hard to understand or to operate.

Usually, a VR application synthesized the virtual "world" from a set of data. This data set could be predefined by other software tools, such an architectural model, or created at run-time, like for example, the fluid dynamics inside an air-conditioned room. The "worlds" are presented mostly in the form of three-dimensional stereoscopic images, although other perceptual channels such as auditory and tactile could also be provided. Users of the applications interact with the "world" by operating devices such as a wand, data gloves etc. Moreover, position trackers play a very important role in feeding additional input to the system: they provide information about the users' position, orientation or even posture, so

that the system can "know" necessary information of users, therefore update the synthetic scene accordingly.

Accompanied with the rapid developing of VR technology, VR applications are used to solve problems in an increased number of domain areas. Robert J. Stone summarized VR applications in domains of Engineering, Micro- and Nano-technology, Aero and Space Engineering, Ergonomics/Human Factors, Defense, Medicine and Surgery, Heritage, Retail, Education, Database and Scientific Visualization (Stone, 2002). In the Virtual Reality Application Center (VRAC) at ISU, VR applications developed or under development cover the areas of architecture walkthrough, scientific visualization, vehicle simulation, virtual battlefield, distributed virtual assembly, etc. (VRAC).

Depending on the problems to be solved, the functionality required by applications varies greatly; some are relatively simple, while others are more complicated.

An example of simple VR applications can be found in the domain of architecture or heritage, such as an architecture walkthrough application (Chan, 1999). In such an application, the structure, shape, appearance, decoration details of architectures have been specified and stored in 3D geometric models which in most cases also include textures and lighting information for a more realistic appearance. These models are loaded when the application starts. With the help of input devices, users "navigate" inside and around the models to explore them at different locations and from different angles. During the running of the application, no change is made to the model data. Generally when the size of data is very large, advanced computer graphic techniques are applied to guarantee the systems' performance. One example of these techniques is level of detail, which "involves decreasing the detail of a model or object as it moves away from the viewer" (Wikipedia, 2005). Similar applications include certain, but not all, scientific visualization used for visualizing pre-

generated data models. Other scientific visualization applications may involve more functions as described below.

In addition to navigation and viewing data, applications in engineering design, such as an immersive CAD tool (Bryden, 2003), require the ability of modifying models at run-time. Modifications of data models include changing models' properties, composing larger models from small building blocks and removing unused components etc. The changed model data in such systems normally are persistent since they represent the outcome of users' work. When the amount of predefined components becomes very large, the management of these data should be carefully implemented, so that users can easily browse, search and load the data that they want.

In the domains of training, operation simulation or ergonomics research, a typical feature of the application is to record sequences of users' actions to evaluate their performance in a specific task to evaluate their performance. In some cases, they benefit from special input/output to create more real effects, like data gloves, devices with haptic feedback (Massie, 1994) or a vehicle cab. Sometimes, a virtual character with artificial intelligent is presented to guide the users in training. There are examples of this applications category (Vance, 2003; Gallagher, 1999; Meglan, 1996). One aspect in common is that the response time in such systems is critical to minimize unwanted side effect including disorientation and motion sickness (Kalawsky, 1993).

Distributed VR applications bring a lot more complications to system design, as well as adding appealing features and practical usage. In a distributed VR system, applications running on different computer systems are connected together by network to perform collaborative tasks. Graphical representation of remote users or objects is always presented locally to feel more like a shared workspace. In this kind of applications, models management has special considerations, e.g. real-time data synchronization and concurrent data access.

Therefore how to minimize the latency caused by the network delay needs to be addressed. A distributed visual battlefield (Bernard, 2004) and a distributed virtual assembly (Kim, 2004) are two examples among many others.

Beside general functions and features described previously, most applications also have their own domain-specific functions. For instance, vehicle simulation software has a vehicle dynamics computation module or communicates to a server that provides this service. Another example is that a training application may need functions to dynamically generate training context based on certain rules and the users' performance.

Moreover, non-function issues are not less important in any VR applications. As mentioned before, performance is extremely critical for the quality of interaction and immersion. Another issue is portability. The complexity makes many VR systems costly. Therefore, it is often desired that the software can be developed and evaluated on inexpensive desktop systems and later used on the VR system. It is also valuable porting the same program in many VR hardware platforms. Real-time performance and portability are achieved by adopting suitable software architectures and proper use of various software techniques.

As stated earlier, VR applications cover a broad range of domain areas. It is impossible to enumerate examples for all domains. The examples provided here are for the purpose of helping readers gain a basic understanding of VR applications.

The study of more applications shows that the elements composing VR application software can be abstracted into four groups: Behaviors, Scene Primitives, Devices and Users. It is important to note that there are many ways to abstract elements for VR applications. This abstraction is from the users' perspective, which best fits the interest of this research. In the following paragraphs, these four groups of elements are discussed.

Behaviors present how the software use the data from devices and system states to change or to influence the virtual world and other system output, such as files I/O, sockets etc., in other words, the functionality of an application. The behaviors can be categorized into two groups: event-based behaviors and time-based ones. Event-based behaviors can be found in almost all applications, such as navigation, collision detection and manipulation of virtual objects (grabbing, moving etc.). Time-based behaviors are also used widely in many applications providing automation of virtual objects. For example, the rotation of an airplane propeller is implemented by changing the position and orientation of the propeller model periodically. Each behavior takes data from input device or system storage, then processes the data and/or forwards data to others, and finally modifies the system status or changes the models' properties. Behaviors vary greatly depending on the specific applications.

A computer-synthesized "world" is the center of a VR application. It includes a graphical scene and audio elements, and sometime, additional information for other perceptual channels. The graphical scene is composed of a set of individual or related geometrical models as well as many special effects such as light, fog etc. Usually, these components are called nodes of the scene. Similarly a sound element contains the wave data, playback pattern, position and intensity of a sound. Data of these graphical nodes and sound elements are the **Scene Primitive** of a VR application. To handle the scene primitive more efficiently, a framework of data organization called scene graph is defined (Bar-Zeey, 2003). Software packages, such as OpenGL Performer (SGI), OpenSceneGraph (OpenSceneGraph) and OpenSG (OpenSG), have been developed to provide the scene graph functionality.

As described previously, a variety of hardware devices are used in VR systems to provide the system with users' information and actions. To build VR applications, the user should first figure out what devices to use and how to use them to interact with the system. On the

software side, **Devices** data need to be represented. Information of each device includes its initial value and updated value.

Users of VR applications act as a special kind of elements. Some of users' information is actually covered by devices and scene graph. For example, the "position" and "orientation" of the users in the virtual world determine the viewpoint in the scene graph, with the special info partly coming from the tracking devices. However, in some cases, it is worth separating the user as an individual element of VR applications, especially when there are multiple users in an application.

VR Juggler applications

In the early stages of VR technology, applications were built up from scratch. Developers had to take care of every technical aspect in detail. As this technology has matured, VR platforms or libraries emerged to help developers reduce the repetitive work. VR Juggler is among such VR platforms.

Since its first release in 1999, VR Juggler has been used to develop a great number of VR applications, for both academic and commercial purposes. Many of VR application examples referred previously are developed on top of VR Juggler.

From the software engineer perspective, VR Juggler hides the complexity of system configuration and provides a unified software structure. All VR Juggler applications have a similar architecture in terms of major classes and methods. From simple VR applications to integration of advanced interactions and algorithms, VR Juggler offers great flexibility and extensibility in the development and execution of applications. More importantly, it supports multiple Operating Systems (OS) and Virtual Reality hardware platforms.

Applying the Object-Oriented design methodology, the interfaces between input devices and the software are abstracted and encapsulated in VR Juggler. Developers handle input data following a set of consistent rules. The input and output hardware configuration files follow a

well-defined specification. To implement an application, developers need to design classes for their domain-specific functionalities, inheriting several major classes and overriding related methods. And the VR hardware platform setting can be performed when launching the application. Thus the same program can be run on different hardware platform easily.

Along with the increase on number of applications developed on top of VR Juggler, implementation of many basic functions of VR applications, such as navigation and object grabbing, has matured. It is feasible for new applications to reuse these methods to improve the productivity.

However, VR Juggler is a development toolkit that requires programming knowledge and understanding of object-oriented programming. In many VR application domains, the experts on a specific discipline, such as architecture, may not have (or have limited) programming skills. Furthermore, because VR Juggler is a toolkit, it takes time and effort to create an application with it.

End-user Programming

End-user programming is a research area aiming at helping "end-users, who have not necessarily been taught how to write code in conventional programming languages, write computer programs" (Cypher, 1993).

The prevalence of end-user programming has its reason from both the user side and programming technology side.

From the user's perspective, software is developed to solve more or less generic problems, while end-users need it to be more specific. Therefore in many cases, users need to customize the software even if they don't have sufficient programming skills and they cannot afford time and energy to get the skill. On the other hand, comparing to the expressiveness and generality provided by full-fledged programming languages, the users just want to tailor the

functionality in a narrower way. They do not need the full-power that the general programming languages provide.

On the programming technology end, along with the development of programming languages, human cognitive models are studied. Better methodologies for expressing the programming tasks are determined. Applying these methodologies can increase the productivity of programming efforts, flatten the learning curve of programming, and encourage the willingness of learning programming skills. When the programming goals are narrowed down to a specific domain, the similarity among functions makes it possible to abstract the domain specific variations. Thus it is possible to hide the implementation complexity from end-users. End-user programming helps end-users by providing more intuitive programming methodology by hiding unwanted details from users.

Although as the name suggests, end-user programming focuses more on the "end-user", it actually can benefit experienced programmers too. In many cases, intuitive programming methodologies can greatly reduce the time used on learning new tools. Hidden of implementation complexity can save much time when programmers just want to create a prototype or to prove a concept.

The major end-user programming techniques include Visual Programming (VP), Program by Examples (PBE), Domain-specific Languages, and Natural Programming etc. For the interest of this research, Visual Programming and Domain-specific Languages are studied.

Visual programming

Visual programming is a very broad research area and there are many definitions of it. According to Brad A. Myers, "Visual Programming (VP) refers to any system that allows the user to specify a program in two-(or more)-dimensional fashion" (Myers, 1994). Examples of the multi-dimensional fashion could be diagrams, icons and tables etc. In contrast the conventional textual representation of programs is considered as one-dimensional.

Visual programming can be divided into two areas, the visual programming language and the visual programming environment. The main difference between these two approaches is that the former is a language using visual syntax whereas the later uses a graphic environment to carry out programming of conventional textual languages (Burnett, 1995). In other words, users can achieve their programming goals purely by working with a visual language. But they need to access textual languages to perform their task when working with a visual environment. In most cases, visual programming is used to refer the former. This leads to some arguments that the visual programming environment is not "real" visual programming. Nevertheless, the borderline between these two sub-areas is not always clear. Hybrid approaches can be applied – a visual programming language may have an underneath textual language to provide more functionality. Users perform common jobs using the visual representation, while using the textual language when extended power is needed. This research does not distinguish these two approaches.

Visual programming is based on the idea that the graphical representatives of program primitives are usually more intuitive than their text counterparts. However, this requires the design of the programming language to be performed properly. And the efficiency of virtual programming should be evaluated by further experiment (Blackwell, 2001).

To help designing and evaluating visual programming tools, Green and Petre constructed a cognitive dimensions framework (Green, 1996). This framework contains thirteen issues coherent with each other. These issues are briefly introduced as following:

- **Abstraction gradient**, the first thing a program designer would ask about a new language is: what are the levels of abstraction it offers? Especially, does programming mean an endless abstracting process?

- **Closeness of mapping**, the more obvious the mapping between the domain problem and the program representation, the easier the programming tool is to learn, although this may mean losing generality.
- **Consistency**, a consistent language would allow the user to "guess" the meaning of the unknown portion based the understanding of the known.
- **Diffuseness**, how many visual units are required to present a meaning? It is a balance between efficiency and flexibility.
- **Error-proneness**, easy to use and easy to guess may mean easy to fall into pitfalls of unintentional mistakes. Avoiding error is a practical consideration in the design of the notation.
- **Hard mental operations**, if users need help from external tools to track the information flow, the purpose of this visualization may have been impacted.
- **Hidden dependencies**, if any invisible dependency exists between entities, the language is introducing error and potential failure, at least extra work in revising the design.
- **Premature Commitment**, users would expect to have all supporting information before they need to make decisions in either design or implementation of a program.
- **Progressive evaluation**, it would be very convenient and efficient if the user could execute a partially-complete program Thus he/she does not have to wait until too late to evaluate the work.
- **Role-expressiveness**, many users grasp the concept by recognizing the role of each element in the whole "picture". Therefore it is worthwhile presenting it to users clearly.

- **Secondary notation**, some programming languages allow methods to present extra information beyond the defined semantics, e.g. comments and naming convention in Java.
- **Viscosity**, programming is a recursive process. The work involved when users make even a small change counts to the overall usability of the language.
- **Visibility**, a user-friendly environment would guide the user to navigate programs in a way promoting understanding the content and the internal relationship.

When designing a visual programming language or environment, these issues are to be considered and trade-off may be made.

Domain-specific language

"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" (Deursen, 2000). Domain-specific languages only abstract specific functionalities and elements in problem domain. So they can hide the complex implementation detail of general functionalities. In a domain-specific language, the expression of programming elements and concepts is more familiar to end-users then general programming jargons.

The domain-specific language can be presented either visually or textually. When presented visually, it is overlapped with Visual Programming Language. Thus we only discuss textural domain-specific language here.

A domain-specific language can be defined independently, for example, a script language for communication; it can also be defined using other standards or languages, of which XML is a good example.

XML, stands for the eXtensible Markup Language (XML), is a language specification developed by World Wide Web Consortium, W3C. It is designed to store any kind of structured information, and to enclose or encapsulate information in order to pass it between different computing systems which would otherwise be unable to communicate. W3C also developed XML schema to "express shared vocabularies and allow machines to carry out rules made by people" (W3C).

Because of its power of storing structured information, XML is used in a great range of areas. It is also a very good tool for designing and expressing domain-specific languages. When designing a domain-specific language, a XML schema can be used to express its semantics and syntax. And XML documents can hold the program.

The advantage of using XML in domain-specific language design is that it is a widely accepted standard. Large numbers of tools have been developed for creating and processing XML files. For example, given a XML schema, many XML tools can generate a XML document. There are also numerous commercial and Open Source XML parsers that can be used to read XML file into and write from programs.

In summary, there is no strict distinction among different end-user programming techniques, and often, the combination of more than one technique provides the best solution for end-user programming.

Code Generation Technology

In some cases, when applying the end-user programming techniques described previously, a bridge is needed between the visual language or domain-specific language and the efficient general-purpose language source code. Code generation technology is that bridge.

"Code generation is about writing programs that write programs" (Herrington, 2003). Code generation technology uses program to generate source code of software based on data of higher-level abstraction. The data abstraction represents the application domain

requirement and solution, while the generated general-purpose language source code contains the implementation details.

It is worth noting that in the domain of compilation, the term of code generation is used frequently. But it has different meaning from what we discussed here. Although the underneath idea is similar, the code generation in compilation domain focuses more on very low level code or machine code, while we aim at generating high level langrage programs like C++, Java, from even higher level abstraction.

The programs that generate other programs are called generators. During code generation, the generator inputs abstracted data of target program, applies generation rules, and finally outputs one or a set of code files.

The abstracted data could come from stored files or from the users' input through a GUI. The data could be organized in the same form of target language or structured data objects that contain all important information to construct code units.

The generation rules can be hard-coded in the generator; it can also be loaded from a specification file. The generation operation may be as simple as searching and inserting or replacing strings in the in-memory source code. A more complicated way is to create the whole class including the definition, fields and methods.

The code generation result can be output back to the original input source code files but in most cases, it is written to new files.

Different implementation decisions of input, output methods make much difference on the complicity and efficiency of the generator. Usually the flexibility and extensibility of a generator has higher priority than generation time, because the generation processes are performed much less frequent than the produced program.

The code generation technology benefits both software engineers and the end-users.

For software engineers, code generation enhances the productivity and improves the quality of the code. Generating programs by a program reduces repetitive work. And the generated code has better consistency in respective of unified API and coding style. Furthermore, debugging and design improvement on code base can be applied to all the generated code.

For the end-user, code generation converts intuitive problem-domain abstraction to general programming language. Therefore it supports end-user programming.

Pure Plug-in Architecture

Plug-in architectures are an attractive solution for building modular and extendable applications. In a plug-in architecture, functions or resources are bundled into independent modules called plug-ins. Plug-ins implement well-defined interfaces so that they can be loaded by the "host application" at run-time to add their functionality or resource to the application.

Pure plug-in architectures push the run-time loading approach even further - applications are made entirely of plug-ins. In such architecture, all functionalities of an application are partitioned to plug-ins, so the role of the "hosting application" is reduced to just loading and running plug-ins. An entry plug-in is needed to start application functions. A plug-in can provide an interface for other plug-ins to connect to, which is called the extension point. Other plug-ins can implement functions required by the extension point. This is called an extension. By extensions, one plug-in can connect to multiple plug-ins, which in turn can also connect to other plug-ins. Thus, in theory, unlimited level of plug-ins can compose a very complicated application.

From a development tool developer perspective, organizing a whole application in a pure plug-in architecture provides great flexibility and extensibility. The tool can be developed in an incremental way - plug-ins that implement primary functions can be developed first, then

more and more plug-ins can be made and deployed later to enrich the functionality. In addition, this approach also simplifies the maintenance task - bug-fixed or function improved plug-ins can replace old ones without disturbing other established program.

In summary, this chapter gives necessary background knowledge of four areas that are related to this research, respectively, VR applications, end-user Programming, code generation, and pure plug-in architecture. Developing VR applications is a complex task, but the development process can be simplified using conventional software engineering techniques. End-user Programming and Code Generation technology together can make the development simpler, and pure plug-in architecture can make it extensible.

CHAPTER 3 REQUIREMENT ANALYSIS

The goal of the research presented here is to provide VR application designers with a tool that facilitate the development of their applications. To achieve this goal, VR application designers' knowledge base and the development procedure are analyzed in this chapter, as well as the primitive requirement of such a development tool.

VR Application Designers

VR applications are used to solve problems across a broad range of domains. For each domain area, numerous domain-specific issues need to be studied and addressed. To provide efficient solutions, it is desired that application developers have deep understanding of the domain. In other words, domain experts need to be involved into the development or even fully handle the development. The software engineering experience of these domain experts varies greatly. Some of them may have sufficient programming skills, while others may not. For simplicity, when analyzing designer's knowledge in this thesis, skilled programmers are grouped with other software engineers, while those non-programmers are referred as domain engineers.

To domain engineers, when applying VR technology to their own domains, basic knowledge of VR system is mandatory. They should know what devices available, how to operate the device, what kind of interaction offered by the system, and what data to deal with, and how to map the geometry data to their domain data. However, the underneath architecture of VR applications need not to be clear to them. From their perception, a VR application is composed of the hardware devices that they use, the data that they manage and the interaction allowing them to solve the problem. The concept of the physical devices is easy to understand, because they use them everyday. As long as the device works as their anticipation, they don't care how the device data are transferred between hardware and software. They understand the basic VR functionalities such as navigation, grabbing,

collision detection, but their understanding is limited at a certain abstraction level. For example, for navigation, they know pressing one button to accelerate and pressing the other to stop. They don't have to know how the function is implemented. Overall, the basic knowledge of VR applications is easy to gain for domain engineers.

In contrast, skilled software engineers deem applications as a set of functional code units. Because most of today's VR applications are developed using Object-Oriented approach, these code units are represented as classes, objects and methods. Due to the nature of object-oriented design, sometimes, the mapping between code units and the physical devices or domain concepts are close. For example, a device with on/off data option may be abstracted as a digital device. However, a lot more classes need to be involved to make the program fully functional. These classes are not necessarily able to directly map to any physical or domain concepts. Software engineers certainly understand the domain engineers' perception of VR elements, but for development purpose, they take application elements in different approaches.

To benefit a wide range of designers, the application elements presented in a developing tool should be close to the domain engineers' knowledge base.

The Development

VR technology develops rapidly. New techniques keep emerging. These techniques include novel input devices and interaction methods, advanced computer technology that can improve the quality of immersion or provide more functions.

As of applying new techniques in other computer areas, when researchers start to introduce an advanced technique in applications, they try a variety of approaches to integrate this technology into existing application frameworks. If they fail, one or more existing frameworks may be extended or modified. Or a novel framework is designed and implemented. After more and more applications are developed, good strategies for applying

become clear. One or more relatively suitable solutions appear. If other applications can reuse these solutions, the productivity of development will be greatly boosted. Thus this research highly emphasizes reusing mature VR application components.

When starting the development, designers first study the domain problems and analyze the requirements. At the same time model data are generated, usually by modeling software such as MultiGen Creator (MultiGen-Paradigm) or 3DS MAX (Autodesk). At this stage they may have the whole data set for constructing the scene or just some data samples for evaluation.

With the deep understanding of requirements and a set of data models in hands, designers choose proper input devices to achieve the best interaction result. Then they start to build up an application "skeleton" to load models and to render them in the virtual environment. Finally they implement interactions. If existing solutions fit in their problems, they can acquire the existing code and adapt it. Otherwise, a set of new classes are created on top of VR libraries. Based on this skeleton, an iterative procedure is performed to customize the interaction and to enrich the functionalities. This is where the most workload of VR application development lies.

In summary, choosing proper interaction devices, constructing an application skeleton to load and to render data, and implementing functionalities to solve domain problems, are the three main tasks of VR application development. Among them implementation of various functionalities is the most time-consuming part of application development. Hence this is the main focus of this research.

The Requirement of Development Tools

The complexity and performance-critical nature of VR applications implies that a high efficient programming language, such as C++, is needed to handle the implementation of VR functionalities. Nevertheless, study of end-user programming and code generation exposes

various approaches of abstracting and presenting domain problems in higher-level languages. Combining these two aspects, a practical solution for VR application developing tools is a multi-layer programming architecture. The top layer is a visual programming environment, which provides graphical representation of VR application elements and visual interaction methods for authoring VR applications. For simplicity, in this thesis, the graphical presentation of application elements is referred as a visual language, although it may not be a fully defined language. Under the visual layer, textural domain-specific language is used for storing the application data for the visual language. And finally, a code generator is used to create C++ source code which is on top of VR libraries.

To be really valuable for VR application development, the design of such a software tool must take four factors into account: simplicity and intuition, flexibility, extensibility, and portability.

Simplicity and intuition

To help diverse application designers, especially those domain engineers described previously, the visual environment must be easy to learn and to use. The interaction of the environment must be straightforward, as well as the visual language elements must be intuitive.

Fortunately computer software has been widely used not only in almost all working environments but also in daily life. Mainstream interaction methods of graphical user interface (GUI), such as double click, drag and drop using a mouse, are mastered by almost all computer users. And the libraries providing these functions have become standard modules of most programming platforms, such as J2SE and Microsoft .NET. It is valuable yet relatively easy to implement these functions in developing tools.

However, making the visual language intuitive is more than that. Applying Green's cognitive dimensions described in the previous chapter, the following requirements should be satisfied:

- The mapping between visual programming elements and domain engineers' cognition of application components should be obvious.
- The granularity of the application abstraction should be preformed at proper levels, so that as less as implementation details are exposed, while as much as domain features can be customized.
- The syntax and semantics of the visual language should be clear and plain, so that users can easily remember it and easily figure out the meaning of an instance of the presentation.
- All necessary information should be presented to users for decision making. If it is impossible to present all information in a concise way, high priority data should be given directly to users while means provided for them to explore the remaining part. Usually, the high priority data indicate those helping users to understand the "whole picture" of the application.
- The relationship between two visual programming units should be presented explicitly. There should be no hidden dependency.

Flexibility

In addition to simplicity and intuition, to benefit as many as VR application designers, a development tool must be also flexible. The flexibility issue here indicates the capability that not only provides domain users with the easy-to-learn functionalities, but also gives advanced users means to implement complex functions and higher qualified code.

When applying end-user programming technology, some of implementation details are hidden. This simplifies the development. But it also conceals the opportunity for advanced implementation. To access this opportunity, the advanced user should be able to access the underneath general-purpose code directly. To prevent this kind of access from compromising the effort of visual programming, special mechanism should be designed. For example, adding save-zone in C++ source code can provide advanced users with certain power to implement complex functionalities while keeping the visual programming system work properly.

Extensibility

As described previously, the VR technology is developing rapidly. New techniques keep emerging. It never stops that some of them become mature and ready to be reused. Thus, the representation of VR applications should base on an open-end architecture. Such architecture provides general interfaces for new functional modules, so that they can easily integrate into the existing program. However, it is possible that the new features can not be integrated unless breaking the underneath software architecture. In this circumstance, the new architecture should be presented in the same way as the original one, in terms of graphical syntax and semantics. For this purpose, the coupling among language layers should be loose.

More than the extensibility of the produced application, the development environment itself needs to be extensible as well. For instance, the development of integrated development environment (IDE) has proved the value of many developing features, such as run time debugging. However, it is unrealistic to include all these nice-to-have features when designing and implementing the visual environment. Therefore if the software architecture of the visual environment is modulated, new features could be integrated later. In Object-Oriented design approach, usually, this goal is achieved by separating the interface and

implementation of each function module, and applying proper design patterns when designing the class or object architectures.

Portability

Original VR applications were running on special-purpose graphic-workstations, which are usually using UNIX operating systems. Today, the capability of the low cost personal computers (PC) has been improved greatly. Especially, when multiple PCs composing a computer cluster, they can provide high computational and graphics power suitable for many VR systems. These PCs or clusters usually work on Windows or Linux operating system. On the VR hardware side, there are platforms such as Head-Mounted Display (HMD) (Bungert, 2005), Cave (Cruz-Neira, 1995) etc. Different platforms fit the solution for different problems.

The diversity of both operating system and VR hardware raises the portability issue in the design of a development tool.

As the software used for authoring other software, the VR development tool has two portability aspects. First, the tool itself should be able to port to different platforms. Second, the application built by this tool should be portable to different OS platforms or VR hardware platforms. It would be very helpful if the tool can generate code for multiple platforms when it is running on a specific platform.

The cross-platform feature of the tool itself can be achieved by utilizing a cross-platform language in development. While the cross-platform issue for the produced program relies on the functionality of the code generator.

In summary, to build up useful development tools, the developers' knowledge should be considered, as well as the development process. The tools should be easy to learn and to use,

flexible for serving both domain engineers and software engineers, extendable for adapting new techniques, and should have good portability.

Based on this discussion, our research focuses on the development of a Java-based visual language framework for the development of VR applications. Our framework specifically addresses intuition of the visual language, portability between operating systems, and extensibility.

CHAPTER 4 PREVIOUS WORK

With the requirement analysis in mind, we review previous efforts and existing VR application authoring tools. Strengths and limitations of them are analyzed, which provide ideas and inspiration for our research. In this chapter, the most popular three commercial and three academic VR authoring tools are discussed.

EON Studio

EON Studio is an interactive 3D application development tool provided by EON Reality. The main focus of this software is on interactive 3D web applications or offline business demonstrations. When working with an additional package, EON Immersive, this software can be used to build virtual reality applications. Users of EON Studio can construct the application by the drag-and-drop approach and filling up tables. The applications developed by EON Reality's software are stored and delivered in files in EON Reality's own format (EON).

Description

EON Studio organizes a VR application as a hierarchical simulation tree composed of folders and nodes.

Nodes are the basic elements of an application. A node could be a visual primitive such as a geometry model, a light, a shape, etc. It could also represent object behaviors such as rotation, walk, gravitation etc. The input devices of applications are categorized into the sensor nodes group. Examples include ClickSensor node, KeyboardSensor node etc. Each node has a set of properties which can be customized in the properties window. The properties of a node vary depending on the node type. A node can contain other nodes. In this case, the container is called a parent node or a frame node, while other nodes are called children nodes. EON studio provides about a hundred predefined nodes which are categorized in to libraries. Users can browse these nodes in the components window.

There is yet another kind of nodes called prototype. Each prototype is actually a sub-tree that contains a group of nodes to perform complex functions. Predefined prototypes are also arranged into libraries. The use of predefined prototype can save user's time and efforts from rebuilding common functional modules.

When building an application, the user drags nodes from the components window and drops them at the proper position in the simulation tree. The interactions among nodes are defined as routes, which are represented graphically. In a route, nodes are presented as an icon on top of two smaller icons. One small icon represents in-fields, the other represents out-fields. One or many lines connect out-fields of one node to in-fields of another. Each line represents as an event. Routes are edited in the routes window in parallel with the simulation tree. A node in a simulation tree does not influence the application unless it is connected to other nodes in a route. Routes in an application are organized into layers.

It is worth noting that the concepts in EON software, such as simulation tree, node, field, route etc, are similar to those in VRML, which is a widely used Web 3D language. For people who are familiar with VRML, EON technology is very easy to understand.

To implement more complex functions, users can use either VBScript or JScript to build script nodes, which can be used in the simulation tree as other nodes. For experienced programmers, EON Reality provides a Microsoft Visual C++ based programming API called EON SDK. By utilizing EON SDK, users can create new node types to perform advanced tasks.

The applications built by EON Studio is stored and delivered in EON's own file formats. Users then use players or viewers to run the application file. With the support of EON Immersive, these applications can run on PC cluster based VR systems.

Strengths

EON Studio has a visual programming environment that is straightforward to learn and used by users without programming experience. The node libraries delivered with the EON software package provide rich functional modules for building applications. Scripting languages and the C++ SDK package support advanced extension.

Limitations

EON software works only on Windows platform although the company has mentioned their interest in extending it to Linux in the next year or two. Applications cannot be ported to VR systems running on high-end computers and those non-Windows PC clusters. Distributed VR applications are not supported. The application must be delivered as data files and played back using EON player and viewer, which also limits the use of the applications.

Virtools Dev

Virtools Dev is a behavior-based interactive 3D development tool developed by Virtools (Virtools). Virtools Dev focuses mainly on Web3D and game development. Yet Virtools also offers an additional software package, Virtools VR pack, to work with Virtools Dev for development of VR applications. In Virtools Dev, visual programming is the major development method. The most notable feature of Virtools Dev is that the system has a 3D layout window to display the structure and run-time effect of the "virtual world". Applications developed by Virtools Dev can be delivered as either data files or standalone executables.

Description

An interactive 3D application is referred as a composition in Virtools Dev. A composition is constructed from two types of elements: data resources and Building Blocks.

Data resources are those media data created by other software. For example, 3D model, sound, image etc. A special data resource type is the character, which is a set of geometrical

models with defined animations. Characters can be created by software such as 3D MAX Studio and exported to Virtools-supported format. Virtools delivers a default set of data resources with the software package. Users can create new data resource sets or import their own data to existing ones. To import external data, users just need to copy media files to proper folders.

Building Blocks (BBs), also called Behavior Building Blocks, represent the behaviors that should be attached to data resource elements or other BBs. BBs are the most important elements in a composition. Virtools provides several hundreds of BBs, which are categorized into a tree structure.

The development environment has four main windows for visual programming, which include a BBs/data window, a runtime 3D layout window, a level manager window and a schematic window. The BBs/data window is for managing BBs and data resources. The 3D layout window is for managing visible application elements. The level manager window displays all elements of an application as a tree structure. The schematic window is used for graphically display the details of BBs' relationship.

The data underneath a composition form a tree structure. The root of a tree is called a level, which contains all entities needed in an application. A level has one or more scenes which determine what can be rendered each time. There are also concepts of places and groups for helping organizing the "virtual world".

When constructing a composition, the user drags data entities from the data resources window, and drops them to the 3D layout window. Then, she selects and drags BBs from the BBs window and drops them onto the proper entity in the 3D layout window. The 3D layout window and the level manager window are synchronized automatically. The user can also drop BBs into the level manager window directly. Sometimes, this is a better way to add BBs because the entities onto which BBs should be attached may not be easily located in the 3D

layout window. For each scene entity, attributes can be set by right clicking the entity in either the 3D layout window or the level manager window, and choosing proper items from the popup menu, for example, "Material Setup" for a 3D object.

When attaching BBs onto an entity, the visual schematic for them is generated automatically in the schematic window. In a schematic, BBs are presented as boxes with ports. A BB can have behavior ports and parameter ports. The connection between behavior ports presents the control flow of the functions, while the connection between parameter ports indicates the data flow. The connection among BBs can be performed in the schematic window.

Users can define behaviors that are composed of one to many BBs. In Virtools, this user-defined behavior is referred as a behavior group (BG). Users can define what ports can be exposed to external behaviors, so that a BG can be viewed and used as a normal BB. The use of BG can simplify the representation of an application, so that the user can browse the functionality of a complicated schematic more easily.

If existing BBs are not sufficient for the application's task, the user can use Virtools Scripting Language (VSL) to create her own BBs. The software contains both a text editor and a debugger for the use of VSL. Furthermore, if VSL still does not satisfy users' need, Virtools SDK, a C++ API can be used to create BBs with advanced functions.

The applications developed using Virtools software can be stored and delivered as Virtools specified data files or standalone executables. When developing immersive VR applications, the VR device is configured using configuration files, so that the development can be performed in computers that are not connected to VR hardware.

Strengths

Virtools software's visual programming environment supports the development by diverse developers. The underneath application structure is presented and can be edited in

multiple views, which makes users understand the development more easily and gives them more power to customize applications. The 3D layout is very intuitive for building up the virtual world. The ability of creating standalone executables and configuring VR devices by files makes it more flexible and portable.

Limitations

Virtools software is Windows-based software, therefore can not be used in VR systems running on other OS. And according to Virtools software documentation, it is very likely that it doesn't support distributed VR application.

WorldUp

WorldUp is an interactive 3D and VR application development tool provided by Sense8 Inc (Sense8). This software focuses on providing comprehensive environment for VR application development. The integrated development environment of WorldUp includes a runtime 3D layout window, a tabbed project panel and other panels and windows. WorldUp supports almost all VR hardware in the market. When working with Sense8's World2World, distributed VR applications can also be developed.

Description

WorldUp uses projects to organize VR applications. A project is organized into a scene graph and a behavior tree. The elements of projects include scene nodes and behaviors.

Scene nodes are the building blocks for the scene graph. WorldUp supports three classes of scene nodes: Graphical nodes, Attribute nodes and Organizational nodes. Graphical nodes are those that can be seen in the 3D layout, such as a shape or an imported model. Attribute nodes can affect the appearance of graphical objects, for instance, fog, light, etc. Organizational nodes, such as a group and a switcher, are used to organize the scene graph in a more efficient way.

Behaviors of a VR project are organized in a behavior scheduler. Behaviors in WorldUp are categorized into two groups, triggers and actions. Triggers are those behaviors that lead to a change of object status. Actions are the behaviors that respond to triggers.

The two main user interface components of WorldUp are the project panel and the development window. The project panel is a tabbed panel with four tabs. In WorldUp, each tab is called a Workview. The four Workviews in project panel are Scene, Model, Behavior and Type Workview. Each Workview is a combination of several reused panes. The development window is to display the 3D effect of scene graph construction.

When building VR applications, a user can use Scene Workview to construct the scene graph. She drags nodes from Nodes pane and drops it into Scene Graph pane. The data in both panes are organized as tree structures. The user can also edit the properties of scene nodes in Property pane, in which data are presented in tables.

The user uses Model Workview to import models. She can select a model in Imported Model pane. Then the 3D appearance of the model shows up in Preview pane. Finally she can drag a model and drop it into Scene Graph pane.

As the name indicates, Behavior Workview is where the user organizes the behaviors of a project. WorldUp does not provide a block-and-connection type of presentation for behavior customization. The user should drag a behavior from Behavior pane or a scene node from Scene Graph pane, and drop the item into Task Scheduler pane. In Task Scheduler pane, the relationship among scene nodes and behaviors can be adjusted by the drag-and-drop approach. Furthermore, the user can also edit properties of any item in Property pane.

Type Workview is used for creating custom subtypes. It contains Type pane and Property pane. Type pane displays a tree structure organizing the complete set of WorldUp types, including non-node types. Sound objects and VR devices are customized here.

WorldUp uses Basic Script as the simple way for extending behaviors function. It ships a C-based SDK library package for advanced users to create behavior objects.

The projects of WorldUp are saved in data files and need players to run.

Strengths

WorldUp is based on a widely used VR development library, World Toolkits. It can support a broad range of VR hardware. Furthermore, it supports development of distributed VR applications. The run-time 3D display is very intuitive for constructing the scene. Predefined behaviors simplify the development and increase the productivity.

Limitations

Although World Toolkit is a cross-platform package, WorldUp requires Windows system as OS platform. This limits the portability. The lack of graphical representation of behaviors makes it a little difficult to track the control flow. Furthermore, Sense8 was sold to a larger company, which is currently maintaining Worldtoolkit, but it does not seem the new company is interested in continuing expanding its capabilities.

InTml

InTml, Interaction Techniques Markup Language, is a research work performed by Pablo Figueroa at University of Alberta (2002). It is not a software development tool, but a domain-specific language for describing all technical features of a VR application. The purpose of InTml is to build a foundation for the development of VR authoring tools.

Description

In InTml, a VR application is composed of four kinds of units: VR objects, object holders, devices and filters. A VR object is an object in the scene that can be seen, touched or heard. An object holder is a container of a VR object that makes it easy to handle. A device represents the data abstraction of a physical device. A filter represents a function module. Each unit has input ports and output ports for interacting with others. The system level

parameters can also be set. Although diagrams are used in InTml document, the language itself does not specify the visual representation of VR applications.

The specification of InTml is defined in DTD files. A VR application is stored in a XML file. Users can use standalone XML tools to validate the application file. Implementation is needed to convert the XML file to an actual VR application.

Strengths

InTml has strong emphasis on interaction techniques. It abstracts a components-driven VR application development structure. The research in this thesis is enriched by some details of the units' definition.

Limitations

InTml is just a specification. There is no implementation available to actually develop VR applications. And the abstraction of VR applications needs improvement to support better scene graph construction.

XJL

XJL is a research work performed by Timothy Griepp at ISU. This research results in another XML based language for description of VR applications. An interpreter-like application is developed to work on top of this language to perform user-specified VR functions (Griepp, 2001).

Description

XJL decomposes VR applications as a set of elements. These elements are categorized into eight groups. Each element has attributes to customize. Thus the description of the functionality of a VR application is represented as a set of elements with their attributes. The representation of elements and attributes in the XML file are simple and clear. This makes it possible for domain engineers to understand and modify it. In this way, the development of VR applications is simplified.

To evaluate the efficiency, the research comes out with a VR application, XJ Nav. This interpreter-like program implements all functions defined by XJL, and performs a portion of them according to the XJL file loaded at runtime.

Strengths

This research is based on VR Juggler, which leads to the potential of building cross-platform and flexible VR applications. Based on the study of previous VR researches, XJL decomposes VR applications into eight groups of elements. This decomposition also inspires this research.

Limitations

VR Juggler is not a component-driven structure. Thus the interpreter approach does not achieve a highly efficient result.

Template Application Selections

Template Application Selections (TAPP) is a set of VR Juggler applications developed by Christopher Just (2004) at ISU. TAPP implements a set of key functions for general VR applications. It can be used as a start point for construction of VR Juggler-based application framework.

Description

As a VR Juggler application, TAPP takes advantage of the cross-platform and run-time configuration features of VR Juggler. It can be easily ported to most operating systems as well as many different VR hardware platforms. In TAPP, the implementation of the scene graph loading functions makes it possible to organize the scene graph in an XML-based configuration file. By customizing this configuration file, users load their own data and perform their tasks with a set of functionalities. Furthermore, all VR Juggler applications use this configuration file approach to specify the hardware setting at run time. As long as the configuration file is built properly, the same program can be run on different hardware setup

without modification on code and recompilation. TAPP has also derived a distributed VR application.

Strength

TAPP is operation system independent and VR hardware independent. The functions that it contains are common for many VR applications. It can support distributed VR applications.

Limitations

TAPP is not a development tool. To add user specific functions, the user must work on C++ level programming. Moreover, TAPP is not component-based software, several behavior functions are developed tightly coupled.

In summary, the review of both commercial software and academic works has shown the valuable features of VR application development tools, as well as the potential improvement space. Most commercial authoring tools have an integrated environment for development. They have a large amount of pre-built building block libraries that are shipped with them to greatly speed up the development process. However, OS and VR hardware portability requirement is not satisfied, as most tools support only the Windows operating system. On the other hand, those research works, although without full functions, have served as an inspiration and starting point for the research presented in this thesis.

CHAPTER 5 SYSTEM DESIGN

Analysis of VR application authoring tools and related work indicates that there is still gap between the requirement of VR application development and features offered by available tools. The most noticeable issue is the limitation on the application portability, such as the multiple-platforms support and VR hardware configurations.

This chapter outlines the design of a VR application development system, the Virtual Reality Application Editor (VRAE). This system utilizes markup languages to describe and store VR applications structure and features. It contains a Java-based visual programming tool, VRAEditor, which is developed using pure plug-in architecture. VRAEditor converts the markup language into graphical representation, from which end-users can then customize VR applications with common computer interaction methods, such as double clicking and drag-and-drop. Finally, VRAEditor can generate VR application source code based on users' work. This design emphasizes the use of source code generation in VR applications development. The source code generated by the default implementation is built upon the cross-platform library, VR Juggler.

The entire design task can be divided into the following three phases:

- To determine the subsystems of the overall development architecture
- To define the interfaces among sub-systems
- To establish the blueprint of the default implementation

The Overall Development System

Based on the requirement analysis and the review of previous work, this research defines a domain-specific language to abstract application elements and attributes. This language includes text-based files to store application information details. The system reuses existing VR research achievements and decomposes VR applications into the framework and

components. When designing the visual programming software for composing and customizing VR applications from components, modularity and reuse are also emphasized. Extensibility, flexibility and portability are taken into consideration during the whole system design process.

The system design is shown in Figure 5-1. As shown in the diagram, the VRAE system contains four sub-systems, which are organized in a two–tier structure. In the lower layer is the language specification, which acts as the base of the whole system. The other three sub-systems, the application code base, the language implementation, and a visual development tool, are sitting on top of the specification. They compose the implementation layer. The arrows in the diagram indicate the dependency among the sub-systems. Details of each sub-system are described as follows:

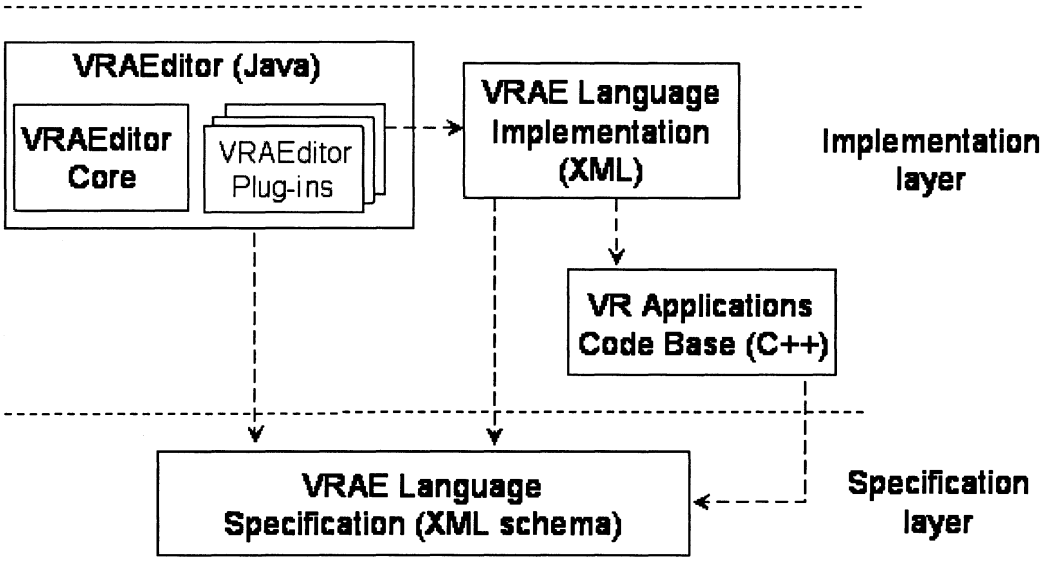

Figure 5-1. VRAE system overview

VRAE language specification

VRAE language specification (mentioned as "the specification" in this chapter) is composed of a set of XML schema definitions (.xsd files) with supplement documents. It defines the structure of elements that carry the VR applications information. These elements

are used either to represent VR applications or to provide code generation data. The specification can be considered as definition of syntax of VR application development languages.

The core of the specification is the definition of three key elements, namely, framework, component and project.

A framework element is where the information of the generic structure of all applications should be stored. It should provide answers to questions such as: What are the files required by all applications? What are the application-level attributes? How to apply the attribute settings to the files? And where can the components be plugged? It also specifies the components type list and the code generation methods list that it supports.

A component element is where a functional module or a data unit can be encapsulated. Components can be plugged into a framework to build applications. A component should contain the information of what files or resources are needed by it and how these resources can be applied to the application. It should also define the attributes that can be modified by users and the interfaces to other components, which are defined as ports in this specification. A component does not necessarily fit only in one framework. Thus the compatibility issue exists in the relationship between a framework and a component. However, for practical reasons, usually, each component is defined for only one framework.

A project element is used to store the users' customization. It has references to one framework and many components. The customization information includes what components are used, how these components are connected to each other, and how the application-level attributes and component-level attributes are set.

The definition of these three key elements implies that the applications are developed with a component-based approach. But the specification does not define how to modularize the applications, which is the task for VR application code base and the language

implementation. The logical relationship among the three elements is presented in Figure 5-2. To be specific, a prefix, VRA, is added to the name of each element, which stands for Virtual Reality Application.

The specification also defines several sub-elements to compose the key elements. The most notable one is the "binding node" element. This element can be found in both the framework element and the component element. It abstracts a unified interface to bridge the application representation to the final code. More specifically, it contains the resource list and the rules to apply users' customization to the application code.

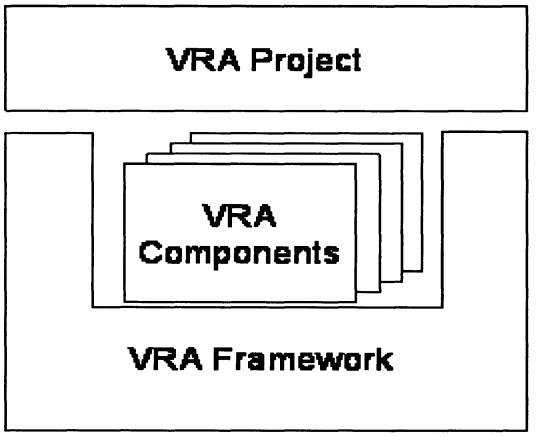

Figure 5-2. Three basic elements defined in the specification

There is another independent element defined by the specification, the "visual definition" element. This element should contain information of how the application is presented visually. Since this element is independent of others, it is possible that applications based on different frameworks apply the same "visual definition". It is also possible to change the visual presentation of one application during the design time.

The data structure of these main data elements are presented in Figure 5-3. More details can be found in the schema definition files.

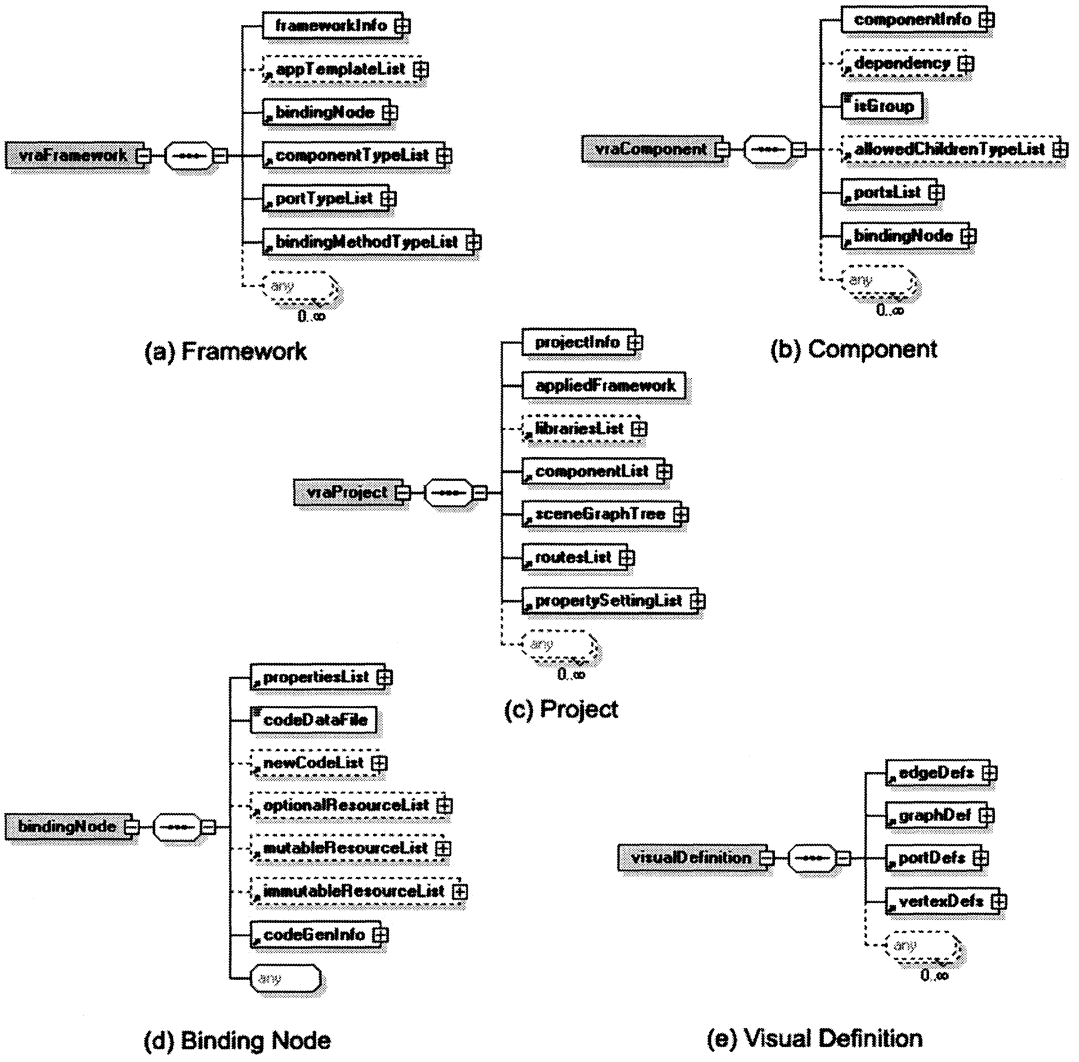

Figure 5-3. Data structure of main elements

VR application code base

A VR application code base is a set of files and data that contain VR application code. It is developed based on successful VR application architectures to reuse previous VR research achievement. The application code base depends on language specification in such a way that it should modularize the functionality of VR applications following the rules implied in the specification. These rules are:

- The common parts of all applications should be encapsulated into a framework.

- Global variables that do not belong to any specific functional module should be presented as application attributes.
- A functional module should be encapsulated into a component.
- Components use input and/or output ports to interact with other components.
- The internal properties of a component should be abstracted as component properties.

The application code base defines the semantics of a VRAE language implementation.

VRAE language implementation

A VRAE Language Implementation is built upon the language specification and a VR application code base. It contains one XML file to describe the framework and many XML files for components, each for one. The number of components depends on how many functions are implemented by the application code base.

The framework and components specify the applications structure and functionality. They are partly hidden from the end users. Users only see the components' name, document, ports and attributes, and framework's attributes. Based on the language implementation, VRA projects can be developed using the visual programming tool, VRAEditor. The project is what users really deal with. Each project element is created in VRAEditor and stored in an XML file for persistency. Final application code set can be generated from the project element.

When building the framework and components from the application code base, the actual code files may be used directly or decomposed and re-organized into separate files. These files are referred in the XML file with additional info about how to modify the code or data and how to integrate them into the application. The code and data files are considered as part of the language implementation therefore should be provided under file trees rooted with the framework.

There are many academic or commercial tools available that can create XML instance against XML schema definition files. Thus the creation of language implementation can take advantage of these tools to apply the language specification.

To maintain a well-organized file structure, it is a good practice to group components into sub-folders based on their type. Each group is referred as a library, and the folder name is the library's name.

The visual programming tool - VRAEditor

VRAEditor is a program that loads VRAE language implementation, presents available components to users, and offers them interaction methods to compose and customize VR applications in an integrated environment. This program is the sub-system that users can interact with. From the users' viewpoint, other subsystems are the underneath platform of this program.

This program is modularized into functional modules. And it defines a set of data types to handle VR application elements. The major elements defined in the language specification have their counterparts in the data type set.

Generic functional modules such as data management and user interaction are implemented only according to the specification. Some other functional modules such as code generation, however, partly depend on the language implementation. But the interface to other part of the program is generalized.

The program presents application data in several ways including tables, trees and graphs. The interaction methods to operate on these data are common to general GUI programs, such as drag-and-drop, double clicking, and filling tables etc. The result of users work is saved into XML files which contain VRA project elements. And when users finish the composition and edition, the code generation functions are triggered and application code is created to the user specified path.

In summary, VRAE is composed of four sub-systems in two tiers. The language specification is in the specification layer. It defines how to decompose existing VR applications to make a modularized application framework, how to present the framework, components and their attributes, and how to finally create VR applications. Other three sub-systems, the code base, the language implementation and the visual programming tool are in the implementation layer. The code base is a set of source code coming from many successful VR applications. They are modified so that they can be composed and represented in the way defined by the specification. The language implementation is a set of XML files that refer to the code base in the form defined by the specification. Finally, the visual programming tool VRAEditor gives users interaction methods to customize VR applications represented by the language implementation and to convert them to C++ based VR application source code.

The Relationship among Sub-systems

As demonstrated in Figure 5-1, dependency exists among the sub-systems. The three sub-systems in the implementation layer rely on the language specification. Within the implementation layer, the language implementation depends on the application code base, and visual programming tool depends on the language implementation. However, these dependencies are presented differently, and the intension of dependency between sub-systems is also different.

The language implementation fully depends on both the specification and the application code base. The later two define the syntax and semantics of the former one. The change in either of the later two will affect the language implementation.

The authoring tool also strongly depends on the specification. It defines several data types corresponding to those XML elements defined by the specification. The functions for loading, storing and outputting these data should be implemented according to the specification.

On the other hand, code base has much weaker dependency relationship with the specification. The development of a VR application code base just needs to follow the rules of modularity, which are easy to follow. When the specification changes, as long as these rules keep unchanged, the code base can keep the same.

The dependency of authoring tool on a language implementation is also limited and weak. Since the tool is modularized, most modules and all the interfaces are built upon the language specification. Only the code generation module might be affected by the language implementation. And this dependency is weak because it is only related to the code generation methods. If new generation methods are added or the whole code generation approach is changed, the code generation module should be modified accordingly. Otherwise, the change of language implementation will not affect the authoring tool.

Decouple VR application representation and code generation

The weak and limited dependency is designed deliberately to decouple the code generation and application representation. This is achieved by grouping code generation related data into the "binding node" element, which is contained in framework and component elements. And a project element does not refer to the binding node data directly. Thus when the VR application code base is improved, it is possible for the users to take advantage of the improvement directly without additional modification. For example, the whole code base may be re-implemented without affecting the project, as long as the components' names and attributes and the framework's attributes are not change. In an extreme case, the code generation approach can switch from generating source code to creating binary binding. As long as a new code generation module is plugged in to take place of the original, the users work can be kept with only minor adjustment. And a totally different application is generated by the authoring tool.

Decouple VR application elements and their visual presentation

Because the visual aspects of components and their relationship, such as color, shape, and line style etc, are defined in the "Visual Definition" element, the coupling between data concepts and the graphical representation is loosed. This element is independent of other elements of a language implementation. The authoring tool can load and apply a different "Visual Definition" at runtime. This offers Human Factors researchers the opportunity to adjust the visual effect to achieve better usability.

The Design of VRAEditor

VRAEditor is developed with a pure plug-in approach by utilizing an open source library Java Plug-in Framework, JPF (Olshansky, 2005). The plug-in level system structure is presented in Figure 5-4. Major plug-in modules include a core plug-in, a data manager, a project manager and a code generator.

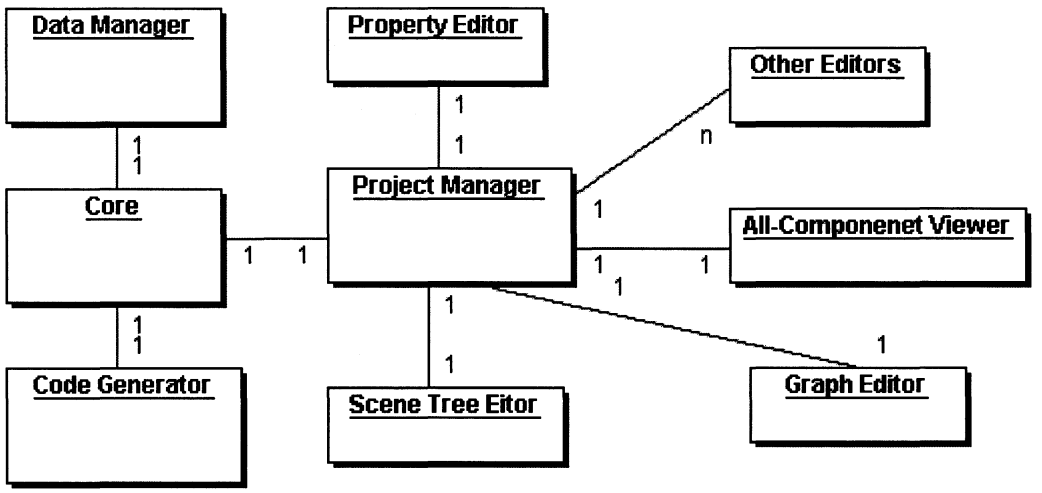

Figure 5-4. Plug-in structure of VRAEditor

Among all the plug-ins, the **core plug-in** is the entry point of the whole system. It loads system setting data, starts other mandatory plug-ins, creates the main frame of the GUI, and translates data among these plug-in modules. It also defines a group of interfaces for all

VRAE specific data types such as VraFramework, VraComponent etc. Other plug-ins can operate on VRAE specific data through these interfaces.

The **data manager** is used to load and manage the framework and component data from XML files, e.g. VRAE language implementation. It also loads or creates project data for other plug-ins. The default implementation classes of VRAE specified data are defined in this plug-in. When the language specification changes significantly, these data type implementations also need modification.

The **code generator** provides functions to validate the authoring work and to generate the final VR application code. The core plug-in assigns project data to the code generator and trigger the code generation through functions defined in the interface "Generator", which is required by the extension point that the code generation extends. The code generator gets information from project data and binding nodes of framework and components, applies generation methods specified in the binding nodes and creates source code in target folders. If new generation methods are defined, it should be implemented in this plug-in. If the whole code generation approach is changed, for example, from source code generation to binary component binding, this plug-in should be replaced by a new one.

The **project manager** is a center module of VR application constructing. It manages and synchronizes loaded projects. It offers extension point "editor" for other plug-ins to edit the projects that it manages. Each editor provides functions to display and edit the project in a special "view", which presents the VR applications features, and offers interaction methods to customize these features. The number of editor plug-ins that can be plugged onto project manager is unbounded. The most notable views that have been implemented are the graph view and scene tree view which are described in the next chapter.

Interactions among plug-ins are performed through carefully designed interfaces. New plug-ins can be added, as well as existing plug-ins replaced by a different implementation, all without affecting other functionalities.

The two-tier structure with four sub-systems in the design of VRAE facilities application representation independency of code generation. From the end-user perspective, both the VR application code base and the programming tools are modularized, so that any part of the system can be modified with minimal impact on other portion. All these design are performed with motivation of making the system extensible and flexible, so that other researchers can easily improve, extend or even reuse part of this research.

CHAPTER 6 IMPLEMENTATION DETAILS

VR Application Representation

VRAE builds VR applications from frameworks and components. A framework provides the code for the skeleton of an application, while components offer the code for the functionality and content of the application.

The framework is not presented to users visually. Currently, the default framework is one developed based on TAPP, which is introduced in Chapter 4. Further details of the default framework are described later in this chapter. More frameworks can always be developed from other well-defined application structures. When multiple frameworks are available, users can choose one that fits their requirements the best.

Components are what users mainly manage in authoring VR application. VRAE defines a component as the data or function unit that can be added onto a framework. A component interacts with others through ports. Each port has its own data type and I/O type. In the default implementation, valid data types include analog, digital, position and command, while the I/O type should be either input or output. When two components interact with each other, their ports should be connected directly or indirectly. However, there are limitations for the connection - only two ports with the same data type can be connected, and the connection must be between one output and one input port.

VRAEditor, the VRAE visual programming tool, provides multiple viewers and editors, e.g. graph editor, property editor etc., for components handling. The interactive relationship among components is presented and edited in the graph editor in VRAEditor. A component in the graph is composed of an icon, a name string and a shape with several ports. Ports on the left side of the shape are input ports, while those on the right side are output ports. Figure 6-1 shows a component with ten ports.

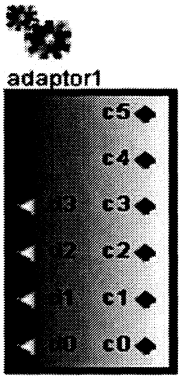

Figure 6-1. A component with ten ports

Components are divided into different groups according to their type. In the default implementation, available components include device, behavior, scene-node, adapter etc.

Device components represent input devices of VR applications. In VRAE, some devices can be directly mapped to a physical device, for example, a *RFWand* or a *Data Glove*, while others may only represent a more abstract concept, such as the *Single Digital* and the *Single Tracker*. The port of a device component is its data unit: for the *RFWand*, a port is a button; while for the *Data Glove*, it represents the analog data of a finger.

Behavior components are those functionality modules that take input data and system states, calculate these data, and finally modify system states or affect scene primitives. Behavior components are the most diverse components that VR applications need. Some of the behavior components provided by VRAE are:

Navigation is an important behavior for VR applications. Users "navigate" in the virtual world so that they can access data models distributed in the "world". Two simple navigation components are provided, namely *Velocity Navigation* and *Inertial Navigation*. These two are similar command-based navigation with different speed control approach.

Grab and Release allows a user to "grab" a virtual object, to move it, and finally, to "release" it to the proper place.

Cycle Group Controller switches the appearance of a set of geometry models.

Navigation Point Recorder records the path of users' navigation and saving it to a file. Users can use it to play back the navigation procedure later.

Scene-node components represent those data units that compose the "virtual world". The most common ones are *Geometry Model*, *Sound* and *Light*, which represent 3D model, sound, and light in the virtual world, respectively.

There are some special scene-node components provided by VRAE to represent a group of models. One of them is the *Cycle Group Node* component. It can contain a set of geometry models -- but at a particular time, only one of them can be rendered. Another component is *Model Reference Group*. It contains a set of references to geometry models. These models have similar features such as being movable, collide-able etc. This component is used when a behavior can only affect models with certain features. For example, the user of a VR application may only grab and move some "hand-tools" rather than a "tree" or a "building".

Adaptor components are used to bridge device components and behavior components that have incompatible port types, so that the users have more choices to control a behavior. For example, a user can trigger a sound by pressing a wand button, or bending a finger. Introduce of adaptor components takes some control logic out of behavior components to make the use of those component more flexible. The control logic of an adaptor component is editable in the property editor. Currently there are two adapter components implemented, a *Digital-Command Adaptor* and a *Mixed Adaptor*. The former one converts digital data to command, while the later one has both digital and command input ports and converts them to command outputs.

It is worth noting that the component shown in a project is actually an instance of that component. The use of the concept instance is because some components, such as most of scene-node components, may be used more than once in a project. When multiple instances of a component appear in a project, to make each instance specific, the name of the instance should be different from that of the component. However, for those components that can only have one instance in a project, their name can be used as the instance name directly. For simplicity, in this thesis, instances are still referred to as components in most places.

Viewers and Editors of a Project

The components and their inter-connection relationship present major portion of the VR application information. However, not all information can be presented in this way. For example, the hierarchical relationship of scene-nodes and the properties of components and the project cannot be presented in the graph editor. Fortunately, the plug-in based structure facilities adding modules to display multiple views of a project. The project manager uses the subject-observer design pattern (Gamma, 1994) to synchronize the change of the project data occurred in different editor plug-ins. Four viewers/editors have been implemented currently for presenting and customizing applications, namely, graph editor, scene tree editor, property editor and all-components viewer.

Graph editor

Graph editor is the main editor plug-in for users to author VR applications. Users can drag components into the graph, connect ports with edges, and double click components to trigger the property edition. Figure 6-2 shows a project graph view in the graph editor.

In the graph editor, components with different types are rendered in different shapes, colors or image patterns. This helps users to easily figure out the component type. The data types of ports are also identified with shapes, which help users chose proper ports to connect. If no proper port is available, adapter components can be used to bridge the gap.

The implementation of the graph editor utilizes an open source Java graph library JGraph (JGraph). Extensions are made to achieve special rendering and operation features required by project authoring, for example, the fixed component-port relationship, and rendering of the port name.

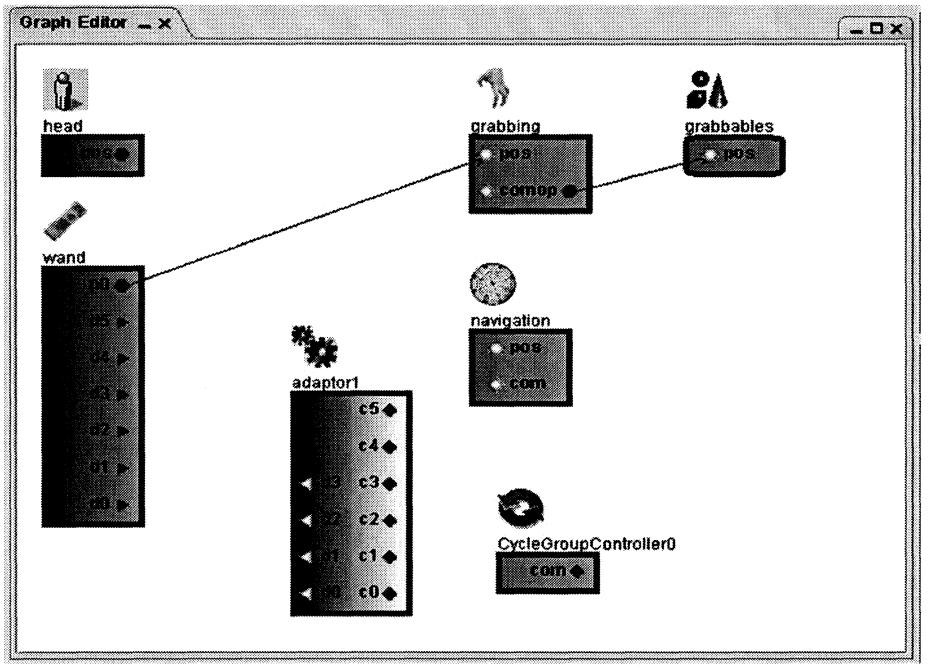

Figure 6-2. A project graph view in the graph editor

Scene tree editor

VRAE uses scene-node components to handle data that compose the virtual world. The hierarchical relationship among these scene nodes is very important in presenting the virtual world. This relationship is presented in the scene tree editor. Figure 6-3 is the screen shot of the scene tree editor.

As the name suggests, scene-nodes are organized and presented in a tree structure. Each single data node such as *Geometry Model*, *Sound* or *Light* can be put into the tree as a leaf node, while group nodes such as *Cycle Group Node* or *Model Reference Group* act as branch nodes of the tree.

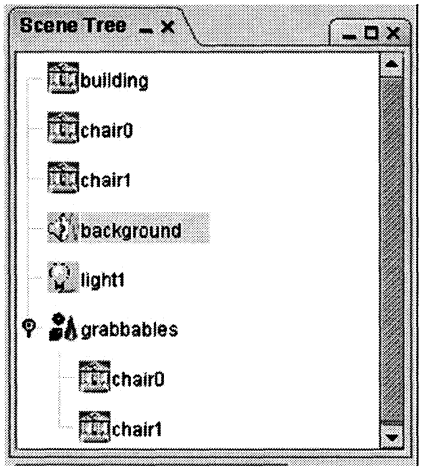

Figure 6-3. Scene tree editor

Since the interaction relationship needs not to be presented in the scene tree, component instances in this view only render the icon and the instance name.

When adding scene-node components to a project, users can drag components from the library and drop them into either the graph editor or the scene tree editor. In the graph editor, not all scene-nodes are used as the target of some behavior's output. For example, a *Model Reference Group* component as a whole is used as the target of *Grab and Release.* Nodes of its children need not to present. To make the graph simple, some scene-nodes can be hidden if they are not connected to others. While in the scene tree editor, all scene-nodes are presented and no other type of components is rendered.

All-components viewer

Because neither the graph editor nor the scene tree editor displays all components of the project, the all-components viewer plug-in is developed to give the user a handy tool to browse all components contained in the project. Components presented in this viewer are rendered as a list, in which each item represents a component instance. The icon and instance name are rendered in the way similar to that in the scene tree editor. Figure 6-4 shows the all-components viewer.

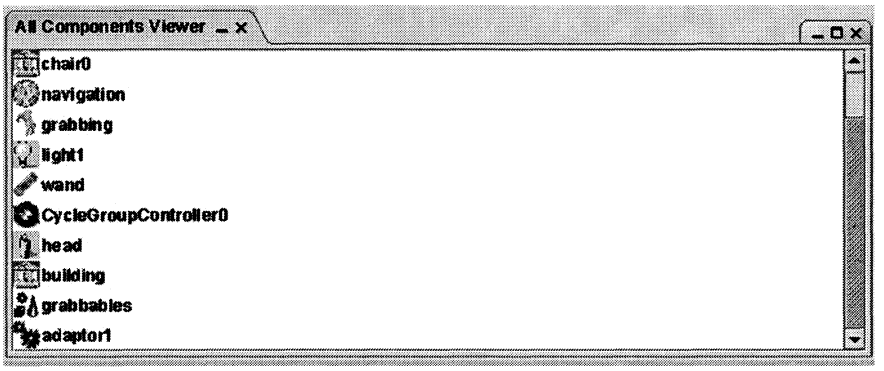

Figure 6-4. All-components viewer

Property editor

When users choose a component by clicking in the graph editor, scene editor or all-components viewer, the property of the selected component is presented in the property editor. Users can edit the properties to customize the application functionality in more detail. For scene-node components, almost all important information, such as the data file path and initial position are set as properties. When no component is selected, properties of the project will be shown in the property editor. Figure 6-5 shows the property window.

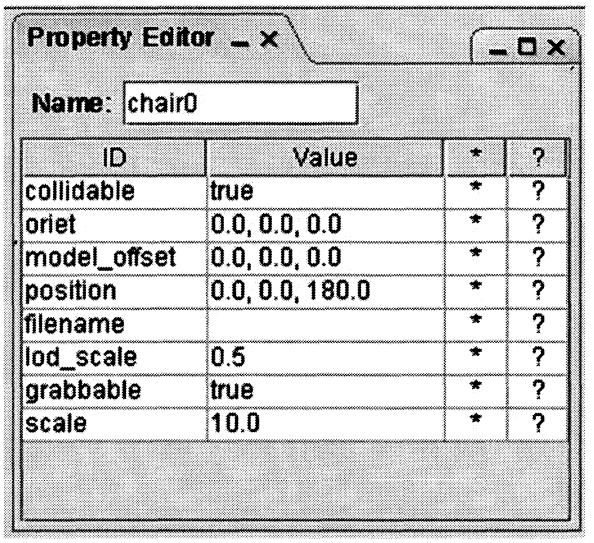

Figure 6-5. Property editor

Property values are presented in the editor as a table. For different property value types, different methods are implemented for editing. In the documentation of Java Swing library, this is referred as table item's *editor* (Sun, 2005).

The property editor has another purpose -- displaying additional information of components. For space reason, the port names in the graph editor use short names (no more than three characters). To help users understand the meaning of each port better, the property editor displays the full name of each port. It also provides help information for each property item.

GUI components of all viewers and editors are organized by using a docking windows system, in which way, the panel for each editor/viewer can be put anywhere in the main frames. And the configuration can be changed at any time. This will help users organize the windows more efficiently. Figure 6-6 demonstrates two different configurations of views. The docking windows system in VRAE utilizes an open source software library, InfoNode docking windows (NNL).

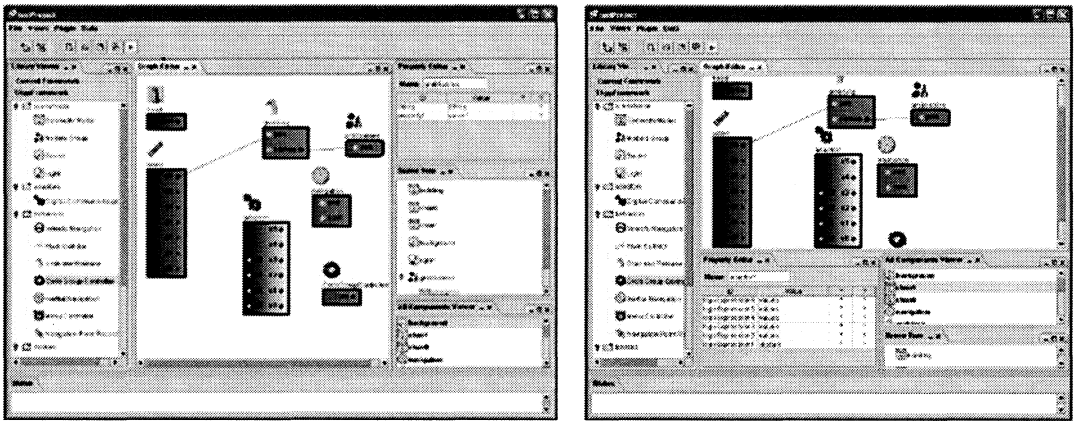

Figure 6-6. Two different configurations of GUI windows

Implementation of Code Generation

The code generator plug-in takes the user-customized project data from the project manager, queries the binding node data from data manager, and creates VR Juggler-based source code to user specified target directory. The generated code contains C++ source code, VR Juggler configuration data files, and other files, such as compilation and execution scripting files etc.

The generation procedure includes tasks of copying immutable code files, creating new code data objects from code templates, modifying code pieces in the code data objects, and writing code to new files. The source code files and code pieces in the code data objects come from the code base. The rules for modifying the code pieces and using them to create final files are specified in the default language implementation, in other words, the default framework and its components.

The default code base

The default VR application code base is developed from the VR Juggler template application selection, TAPP, introduced in Chapter 4. As a general-purpose VR application, TAPP has implemented functions that are essential to all VR applications. These functions include setting and initializing hardware devices, loading models, recording motion paths, and interactions such as navigation, object manipulation and collision detection etc. Although these functions are not implemented in a component-based approach, they have much similarity, in terms of function calling scheme. For example, if a behavior needs data from input devices, it always has initialization function which usually is called in the application class' *init()* function and the main processing function may appear in the *preFrame()* function. This similarity helps decomposing the application into components.

For our research TAPP code is transformed to VRAE default code base. The class structure of TAPP is used as the structure of the framework. This structure includes major

C++ classes such as ViewerApp, Model etc., and other types of code such as the VR Juggler configuration file and compilation script files. During the transformation, some functions are used directly, some are modified and event re-implemented, and the file directory tree of the code is re-organized.

After the transformation, the source code is prepared for creating the framework data files and components data files, in other words, ready to create the default language implementation.

The default language implementation

To create the language implementation, a framework data file and several component data files are built. These files contain references to source code files, listed in four different categories, namely, *code data files*, *immutable source*, *mutable source* and *optional source*.

Some of the source code needs major changes according to the users' customization. These code files are decomposed to many code pieces and organized into *code data files*, which are XML files containing multiple code pieces and their properties.

There are also some codes that do not need any change. They are just copied to proper directories during code generation. These codes are referred as *immutable source*.

Some code need minor modification according to the project data. They are also copied to the target directory during code generation, and the copy will be modified later. These codes are referred as *mutable source*.

Other source code files may or may not be used in the target code depending on the property settings. If used, minor change or no change will happen on them, they are referred as *optional code*.

Source code types diverse greatly for different components. For scene-node components, the code mainly appears in the VR Juggler configuration file. For behavior components, the

code can be as simple as just one piece of code for one function; it could also contain many classes and code pieces, even resource data files. For devices component, VR Juggler configuration files are used, as well as device initialization code and data update code in C++ classes.

Generation of source code

When generating code, the immutable source and mutable source are copied to the target file tree. The code data files are parsed and a set of code data objects are initiated. The code pieces in code data objects are modified according to the property setting and port connection. Code modification information of mutable source is also collected and organized into a map data structure. Finally, new classes and other types of source code file are created and written to the target file tree. Mutable sources are modified as well.

The data files of geometry models or sounds may be copied to the data folder, which is located in the target file tree, so that users can package the whole code tree and use it anywhere. But if the user chooses not to copy, these data will be referred as absolute paths. In this way, hard disk space can be saved, but the portability is compromised.

Generated code also includes files for compilation and execution. If the user chooses Windows system as target, a Visual Studio.NET project file will be generated. If UNIX/Linux is selected, configure file and makefile template will be created. Users can use the Automake facilities to build applications. This utility is open source, and usually available in Unix/Linux system installation. If absence, it can be downloaded and installed freely.

CHAPTER 7 RESULT AND DISCUSSION

To Develop VR Applications Using VRAEditor

With the design and implementation of VRAE, end users can create a VR application without knowing any C++ programming. This chapter presents the end-user-facing VR applications authoring process, specifically, the usage of visual programming tool, VRAEditor.

To launch VRAEditor

VRAEditor is developed fully in Java. It can run on any system that has Java runtime 1.4.1 or later version installed.

When VRAEditor is launched, preference data will be checked, the data manager plug-in will try to find data path to load the framework and components. If no preference data is available, say, when the first time the program is running, the default path will be scanned. If components data are loaded successfully, they will be displayed in a library viewer panel in the form of a simple tree. These components are grouped into libraries for better organization. Figure 7-1 shows the main frame with components displayed in the library viewer.

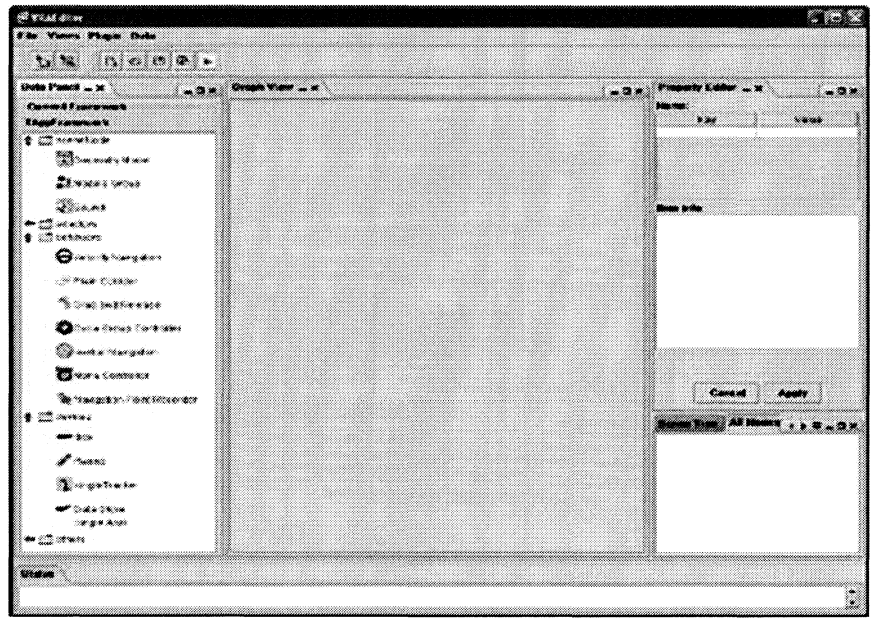

Figure 7-1. Main frame with components in the library viewer

To start a new project

To develop a VR application using VRAEditor, the user either starts from an empty project or chooses a predefined template as the starting point. A template contains a set of basic components such as navigation, tracking devices etc. A portion of the relationship among components, e.g. connections among ports, and default properties of components and the application are also set. This saves the user's time and effort from repeating the general steps.

If the user chooses to create an empty project, a project without components nor connections will be created. The scene tree is also empty. The framework property list will be copied to the project as project properties. Any default value available will be assigned as property value. Figure 7-2 shows the windows with a new empty project.

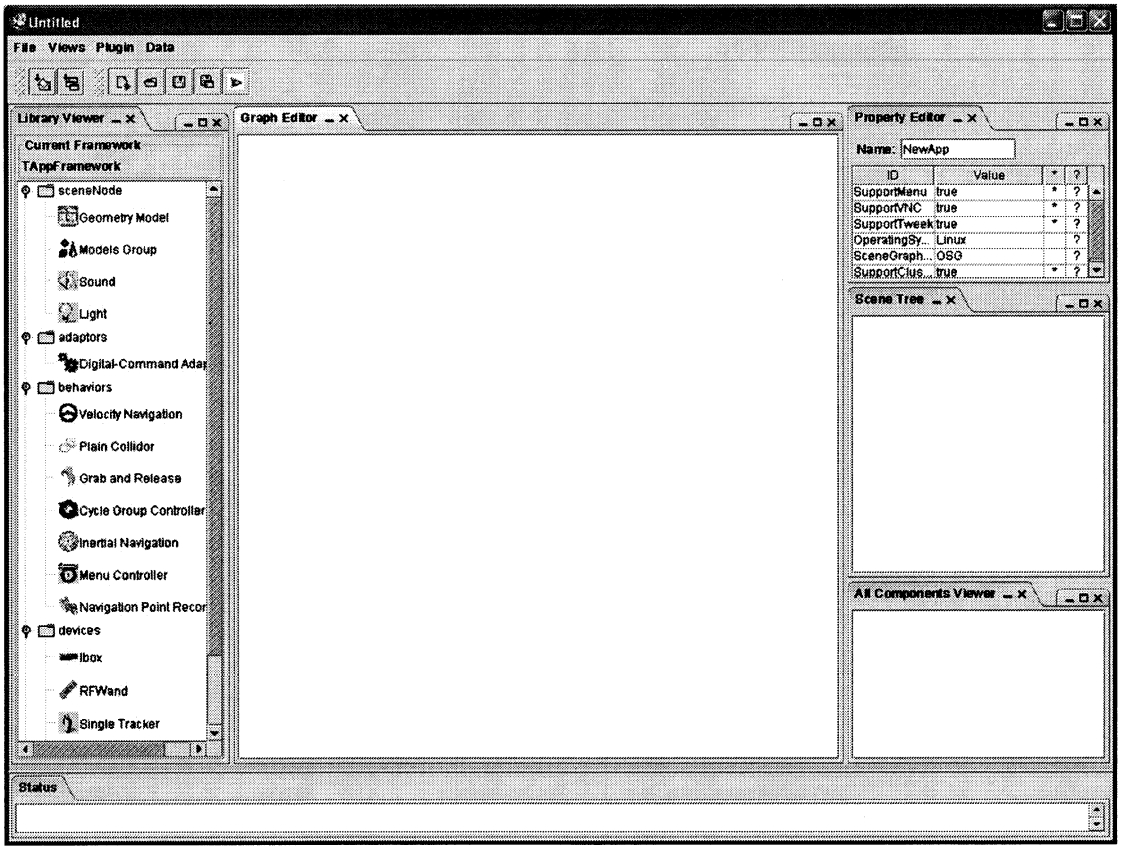

Figure 7-2. Main frame with a new empty project

To build VR applications visually

To customize a VR application, the user needs to add components, to setup connections, and to specify properties value to customize the application. To add a component, the user just need to find it in the library viewer, drag and drop it into the graph editor or the scene tree editor. For the scene tree editor, only scene-node components are accepted. When the components are dropped, a new instance will be added to the project.

Figure 7-3 shows a project with components added. In this figure, two devices are selected: the RFWand and the Single Tracker. The former is a device that contains 6 buttons and one position tracker, while the later is attached to the user's head to detect the user's position and orientation. The behavior components include a Velocity Navigation, Grab and Release and Cycle group controller. Three geometry models are used in the scene, a building and two chairs. The Grabable Group has the two chairs as children. This means that the two chairs can be grabbed and moved by the VR application user when the application is running.

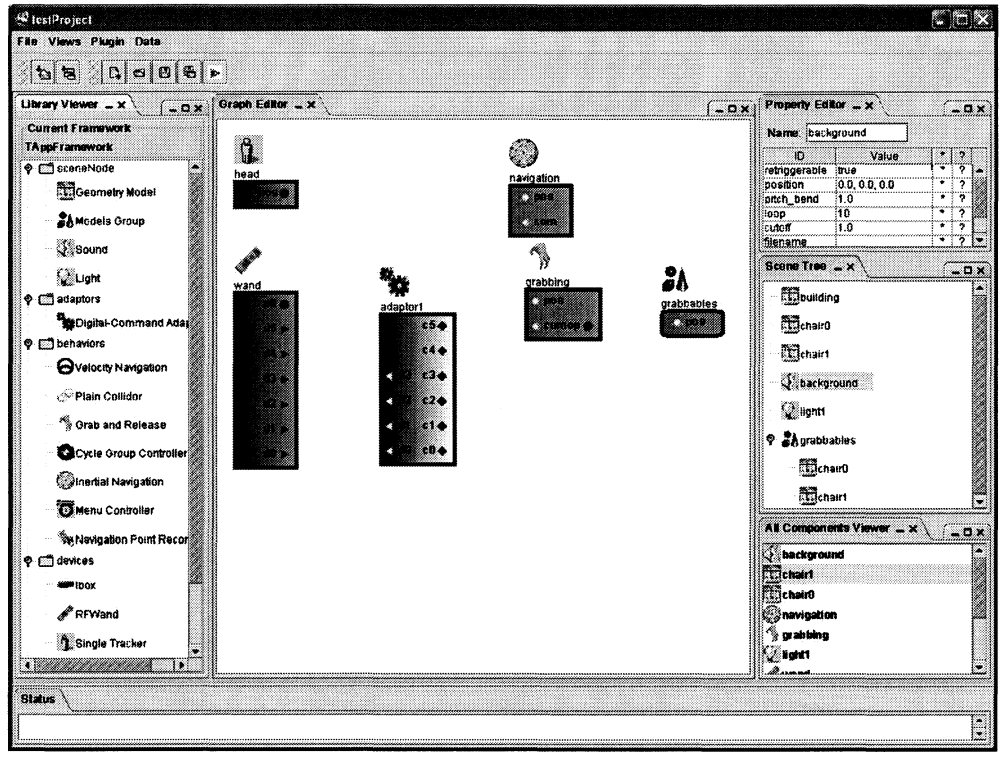

Figure 7-3. A project contains several components

In Figure 7-4, ports are connected with edges, to specify the interaction relationship among components. Because the buttons of RFWand is digital output, they are not compatible with the command input required by navigation and grabbing behaviors. Thus an adaptor component is used, which converts the digital output to command.

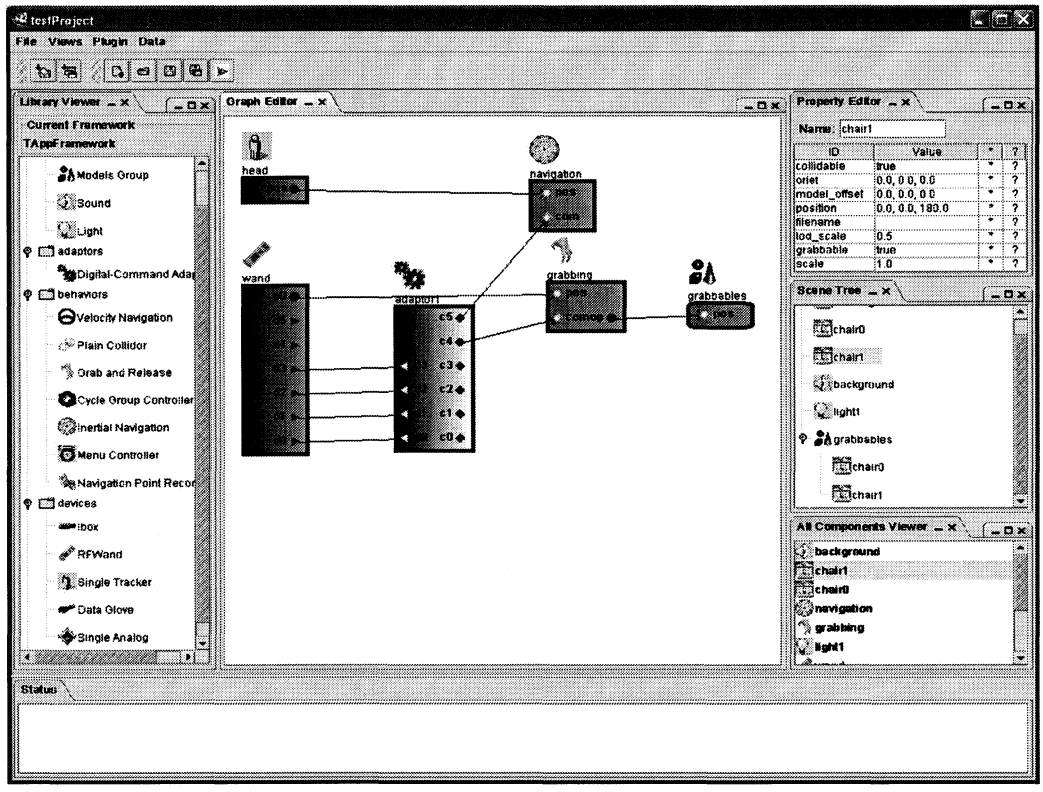

Figure 7-4. A project contains components and connections

Figure 7-5 presents the property of the scene-node "chair1", which is an instance of the component Geometry Model. Users can use the property editor to modify the property setting of selected components.

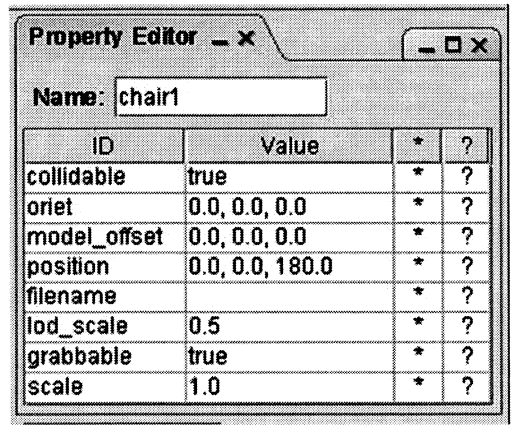

Figure 7-5. Properties of scene-node "chair1"

To generate code

When the application is constructed and set properly, the user can press a button to trigger code generation. The program will check the project completeness and create a set of C++ based source code files to a specified directory. The completeness checking will check three issues:

- The project has at least one device, one behavior and one scene-node model.
- Values of all required properties are set.
- Required ports are connected.

When the code generation process finishes, a set of source code will be created in the target folder. This code is a standard VR Juggler program. The user then compiles and runs this code to perform her domain task.

Discussion

By using VRAEditor, VR application developers build applications from pre-defined components. The users' tasks include selecting proper components, specifying the interaction relationship among components, assigning hierarchical relationship of scene-node components, and customizing properties of components. How well VRAEditor can serve

users for customizing VR applications is determined by the way application information is presented to users and the interaction methods that users can use for the customization. The author makes an attempt to discuss multiple aspects of this topic.

Decomposing VR applications into components provides the opportunity to hide implementation details from users, so that they can focus on the domain problems. In VRAE, the boundary of components is determined as close as possible to the concept unit in the "real world". For example, most of device components have their physical counterpart; a primitive scene-node component represents a dependent data unit, such as a sound file, a 3D model; one behavior component has the functionality for affecting one aspect of the application. This kind of closeness makes users mapping the real world concept to programming components easily. Consequently, the study and use of the software can be simplified. However, it may be hard to represent new concept units by these components. Thus abstracted components are also needed for better flexibility and extensibility. VRAE provides components such as adaptor components, single unit devices, to fill the gap. To provide practical development power, more components need to be provided, and trade-off between abstraction and intuition is needed when developing these components.

Introducing the concept of port divides the interface among components into smaller units. The data types of these ports determine what information can be transferred among components. Current VRAE implementation defines four type of port, namely, digital, analog, position and command. These data types serve the current components well. More data types are expected to be introduced with new components.

Properties of the project and components offer users more details of the application to customize. The property editor arranges property data into tables. This makes reading and editing property values easier. Better table rendering and editing format is still needed to help users understand the meaning of properties and to ensure correctness of users' input.

Multiple views of the project data provide users with more options to customize VR applications. Current editors provide functions required for application authoring. New editors may improve the functionality and make the development more intuitive. Credited to the use of plug-in framework and docking windows libraries, addition of new editors is simplified.

Source code generation is used by VRAE. The final code is generated from code templates in the code base. In this way existing development achievement is reused. Generating source code directly can decrease the use of run-time binding, thus improves the performance of VR applications. Because VR Juggler is cross-platform at the source code level, generating source code can take advantage of the portability of VR Juggler. Additionally, source code generation gives advanced users an opportunity to implement more complicated functionalities based on generated application.

The quality of generated source code greatly relies on the quality of the code base. If there are bugs in the code base, then these bugs will stay in the final code. However, when the code base is improved or bugs are fixed, users can re-generate the source code without modifying the project data. Thus in long term, the source code quality can be improved solidly.

The disadvantage of source code generation is that users have to go through the whole process of generation and compilation. This delays the feedback of users' work. Improving the code generation function and adding functional modules to go through the whole process automatically can decrease the response time for feedback. The later approach is facilitated by the plug-in structure.

The two-tier system architecture distributes VR application development information into four sub-systems. The general rules of decomposing an application and representing functional modules are carried by the specification; the source-code templates are contained in the code base; the language implementation bridges the rules and implementation; and the

Java program reads in all these information, provides users with interaction methods and performs the code generation rules based on those data. VR application elements and code generation data are in data files rather than hard-coded in the Java program. This allows individual improvement of different aspects of the system, thus the system is more flexible and extensible. The disadvantage is that the development of this system is more complicated.

In summary, by using visual programming and code generation, VRAE simplifies VR application authoring. Design of the system framework and the implementation of functionalities provide good extensibility and flexibility. The source code generation approach and using Java to develop the visual programming tool ensure the portability of both the produced VR applications and the tool itself.

CHAPTER 8 CONCLUSION AND FUTURE WORK

Conclusions

This research establishes a framework for developing the visual programming front-end for the widely used open-source library, VR Juggler. The system provides sufficient flexibility and extensibility, as well as portability.

By visual programming, users without sufficient programming skills can easily develop a simple VR application. For experienced programmers, this tool can also save time and effort on prototyping or initializing the infrastructure of a complicated VR application.

Generating source code instead of binding binary components is one of the major features of VRAE. It provides better platform portability and VR hardware portability. In addition, it gives users more control over advanced functionalities. However, users can only preview the feedback of their customization through the whole process of triggering the code generation, then compiling the program and finally running the program outside of the authoring tool. Fortunately, the plug-in based structure makes it possible to add more development tool modules, including automation of the preview process to improve the program response.

As described in previous chapters, the system design eliminates the coupling between the representation of applications and the actual code. The design of the Java tool also decouples the code generation with the application authoring. The isolation makes it easy to adopt new code base when either VR Juggler or the code base itself is improved.

Future Work

Possible future improvement of VRAE can be explored from the two major aspects of this system -- the Java program functionality and the components library.

To implement new plug-in modules

Because the Java program VRAEditor is developed in the plug-in architecture, adding functionalities means developing more plug-in modules. New modules can be used by the project manager, the data manager or the code generator.

For the project manager, a 3D scene viewer is of the most interest. If such a plug-in is provided, users can view the 3D scene output, and adjust the position and orientation of 3D models at development time. That would considerably improve the usability of this tool. To implement this plug-in, a Java-based 3D rendering library is needed. Candidates include Java 3D (Java 3D) and JavaOSG (JavaOSG). Because both libraries have platform-dependent code involved, the platform portability issue need be investigated.

An adapter editor is also desirable for editing adaptation logic of adapters visually. Such an editor uses graph elements to represent primitive logic units such as "and", "or" etc. as well as comparator, such as "equal", "greater than", etc. It can make the customization of adaptors more efficient and less prone to errors.

For the code generator, a source code editor may help advanced users manipulate the generated source code. To edit C++ code, it is better to mark some "safe zone" in the code and users are only allowed to put code in the safe zone. In this way, users' customization is controlled, and will not confuse the program when it is read back. To edit the configuration file, existing tools provided by VR Juggler may be re-implemented as a plug-in of VRAEditor.

To improve the quality of the Java program and to enrich its functionality, refactoring the software and converting it to Eclipse plug-ins is another option. Eclipse is an open source IDE. It has been used in a very broad area. It also uses pure plug-in architecture. Actually, the development of JPF, the plug-in library used in current VRAEditor was inspired by Eclipse and has taken the plug-in architecture introduced in Eclipse as its basic model. Eclipse has

enormous third party plug-ins available, which provide functions valuable for VRAE. Moreover, Eclipse provides tools for plug-in development which can greatly help adding new functionalities. However, refactoring and re-implementation requires a great amount of work, and the system will be more complicated.

To develop and improve component libraries

To make VRAEditor a practical development tool, much more components need to be added to the libraries. All kinds of input devices are needed to provide the VR user better interaction methods with the virtual world. More complicated scene-nodes are needed to build a more realistic virtual world. And a large amount of behaviors can be developed from successful VR applications to support more VR functionality. One example of a behavior component is a physics simulator, which can simulate the physical force action performed on a virtual object. Such a component can be developed by using existing open source physics engines, such as the Open Dynamics Engine (Smith, 2005).

To make the development of components more efficient, software tools are also needed. These tools can help users to parse and analysis the source code, convert them into component data, and set icon to the component.

REFERENCES

Autodesk Media and Entertainment. Autodesk 3ds Max. Retrieved June 3, 2005, from http://www4.discreet.com/3dsmax

Bar-Zeev, A. (2003). *Scenegraphs: Past, Present and Future*. Retrieved June 3, 2005, from http://www.realityprime.com/scenegraph.php

Bernard, J., Cruz-Neira, C., Oliver, J. & Sannier, A. (2004). *Command and Control Embedded Training*. Final Technical Report prepared for Air Force Research Laboratory/IFKE. Retrieved June 3, 2005, from http://www.vrac.iastate.edu/~sannier/Battlespace/

Bierbaum, A. (2000). *VR Juggler – A Virtual Platform for Virtual Reality Application Development*. Master Thesis. Iowa State University, Ames, IA.

Blackwell, A.F., Whitley, K.N., Good, J. & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review 15(1)*, 95-113.

Bryden, K.M., Chess, K.L., & McCorkle, D.S. (2003). Application of Virtual Engineering Tools in Power Plants. P*resented at the 2003 Electric Power Conference*, Houston, TX.

Bungert, C. (2005). *HMD/headset/VR-helmet Comparison Chart*. Retrieved June 3, 2005, from http://www.stereo3d.com/hmd.htm

Burnett, M.M., & McIntyre, D.W. (1995). Visual Programming - Guest Editors' Introduction. *IEEE Computer 28(3)*, 14-16.

Chan, C.S., Maves, J. & Cruz-Neira, C. (1999). An Electronic Library for Teaching Architectural History. *Proceedings of The 4th Conference on Computer Aided Architectural Design Research in Asia* (pp. 335-344). Shanghai Scientific and Technological Literature Publishing House.

Cleaveland, J.C. (2001). *Program Generators with XML and Java*. New Jersey: Prentice Hall Inc.

Cruz-Neira, C., Sandin, D.J., & DeFanti, T.A. (1993). Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *ACM Computer Graphics, 27(2), July 1993,* 135-142.

Cruz-Neira, C. (1995). *Virtual Reality Based on Multiple Projection Screens: The CAVE and Its Applications to Computational Science and Engineering*. Ph.D. Dissertation. University of Illinois at Chicago, Chicago, IL.

Cypher A. (ed.) (1993). *Watch What I Do: Programming by Demonstration*. The MIT Press.

Czarnecki, K., & Eisenecker, U.W. (2000). *Generative Programming, Methods, Tools, and Applications*. Addison-Wesley.

Dai, F. (ed.) (1997). *Virtual Reality for Industrial Applications*. New York: Springer-Verlag Inc.

Deursen, A.V., Klint, P., & Visser. J. (2000). Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices, 35(6), June 2000*, 26-36.

DIVERSE Home page. Retrieved June 3, 2005, from http://diverse.sourceforge.net

Durlach, B.N.I., & Mavor, A.S. (1995). *Virtual Reality: Scientific and Technological Challenges*. Washington, DC: National Academy Press.

EON Reality, Inc. *EON Studio Home page*. Retrieved June 3, 2005, from http://www.eonreality.com/products/eon_studio.htm

FakeSpace Systems. *ImmersaDesk Home page*. Retrieved June 3, 2005, from http://www.fakespace.com/ideskm1.shtml

Figueroa, P., Green, M., & Hoover, H. J. (2002). InTml: A Description Language for VR Applications. *Proceeding of the Seventh International Conference on 3D Web Technology (Web3D'02) (Tempe, Arizona, USA, Feb. 24-28, 2002),* (pp. 53-58). New York: ACM Press.

Gallagher, A.G., McClure, N., McGuigan, J., Crothers, I., & Browning, J. (1998). *Virtual reality training in laparoscopic surgery: A preliminary assessment of minimally invasive surgical trainer virtual reality (MIST*VR). *Endoscopy*, 31(4), 310-313.

Gamma, E., Helm, R., Johnson, R., & Vissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.

George, C.E. (2000). Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models. *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group*, April 10-13, 2000, Corigliano, Calabro, Cosenza, Italy, 33-43.

Green, T.R.G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7, 131-174.

Griepp, T.P. (2001). *XJL-a XML Schema for the Rapid Development of Advanced Synthetic Environments*. Master Thesis. Iowa State University, Ames, IA.

Held, R., & Durlach, N.I. (1991). Telepresence, time delay, and adaptation. In Ellis, S.R. (Ed.), *Pictorial Communication in Virtual and Real Environments (Chapter 14)*. New York: Taylor and Francis.

Herrington, J. (2003). *Code Generation in Action*. Greenwich, CT: Manning Publications.

Java 3D API. Retrieved June 3, 2005, from http://java.sun.com/products/java-media/3D/

JavaOSG Bindings. Retrieved June 3, 2005, from http://www.noodleheaven.net/JavaOSG/javaosg.html

JGraph Home page. Retrieved June 3, 2005, from http://www.jgraph.com/

Just, C. (2004). *VR Juggler Template Application Collection*. Retrieved June 3, 2005, from http://www.vrac.iastate.edu/~cjust/templateweb/templateapps/index.php

Kalawsky, R. (1993). *The Science of Virtual Reality and Virtual Environments*, Addison-Wesley.

Kim, C., & Vance, J.M. (2004). Development of a Networked Haptic Environment in VR to Facilitate Collaborative Design Using Voxmap Pointshell (VPS) Software. *ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Salt Lake City, Utah, September 28 - October 2, 2004, DETC2004/CIE-57648.

Massie, T.H., & Salisbury, K. (1994). The PHANTOM Haptic Interface: A Device for Probing Virtual Objects. *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*. Chicago, IL.

Meglan, D. (1996). Making surgical simulation real. *Computer Graphics*, 30(4), 37–39.

MultiGen-Paradigm. *MultiGen-Paradigm Home page*. Retrieved June 3, 2005, from http://www.multigen.com/

Myers, B.A. (1994). Program Visualization. Marciniak, J.J. (ed.) *Encyclopedia of Software Engineering* (pp. 877-892). New York: John Wiley & Sons.

NNL Technology AB. *InfoNode Home page*. Retrieved June 3, 2005, from http://www.infonode.net/index.html?idw

Olshansky, D. (2005). *Java Plug-in Framework (JPF) Project*. Retrieved June 3, 2005, from http://jpf.sourceforge.net

OpenSceneGraph Home page. Retrieved June 3, 2005, from http://www.openscenegraph.org

OpenSG Home page. Retrieved June 3, 2005, from http://www.opensg.org

PowerWall Home page. Retrieved June 3, 2005, from http://www.lcse.umn.edu/research/powerwall/powerwall.html

Sense8 Corporation. *WorldUp Home page*. Retrieved June 3, 2005, from http://sense8.sierraweb.net/products/wup.html

Sense8 Corporation (1998). *WorldToolKit Release 8 Technical Overview*. Retrieved June 3, 2005, from http://sense8.sierraweb.net/downloads/wtk_tech.pdf

SGI. *OpenGL Performer Home page*. Retrieved June 3, 2005, from http://www.sgi.com/products/software/performer

Shaw, C., Green, M., Liang, J. & Sun, Y. (1993). Decoupled Simulation in Virtual Reality with the MR Toolkit. *ACM Transactions on Information Systems*, 11(3), 287-317.

Smith, R. (2005). *Open Dynamics Engine (ODE).* Retrieved June 3, 2005, from http://www.ode.org/

Stage3 Research Group at Carnegie Mellon University. Alice Home page. Retrieved June 3, 2005, from http://www.alice.org

Stone, R.J. (2002). Applications of Virtual Environments: An Overview. Stanney, K.M. (ed.) *Handbook of Virtual Environments* (Ch42), pp.842. Mahwah, New Jersey: Lawrence Erlbaum Associates.

Stuart, R. (2001). *The Design of Virtual Environments*, 2nd Edition. New Jersey: Barricade Books Inc.

Sun Microsystems, Inc. (2005). *The Java™ Tutorial*. Retrieved June 3, 2005, from http://java.sun.com/docs/books/tutorial/uiswing/components/table.html#editrender

Vance, J.M. (2003). Interactive product development in a virtual environment utilizing haptics. *Proceedings of National Science Foundation Design, Service and Manufacture and Industrial Innovation Grantees and Research Conference*, Birmingham, AL.

Virtools. *Virtools Dev Home page*. Retrieved June 3, 2005, from http://www.virtools.com/solutions/products/virtools_dev.asp

VRAC. *Virtual Reality Application Center research page*. Retrieved June 3, 2005, from http://www.vrac.iastate.edu/research/index.php

W3C. *W3C XML schema Home page*. Retrieved June 3, 2005, from http://www.w3.org/XML/Schema

Wikipedia Foundation. (2005). Level of Detail (Programming). *Wikipedia, the free encyclopedia*. Retrieved June 28, 2005, from http://en.wikipedia.org/wiki/Level_of_Detail